MATEMATICKO-FYZIKÁLNÍ
FAKULTA
Univerzita Karlova

BAKALÁŘSKÁ PRÁCE

Patrik Smelík

Simulátor počítačové hry Gwent

Katedra softwaru a výuky informatiky

Vedoucí bakalářské práce: Mgr. Jakub Gemrot, Ph.D.

Studijní program: Informatika

Studijní obor: Programování a softwarové systémy

Praha 2021

Rád by som poďakoval vedúcemu práce Mgr. Jakubovi Gemrotovi, Ph.D. za to, že mi umožnil spraviť túto prácu, že ma vôbec prijal, keď som hľadal vedúceho na ročníkovú prácu, za všetku pomoc a všetky rady, ktoré mi poskytol a za jeho trpezlivosť, že to so mnou vydržal až do konca.

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona.

V …..... dne..... podpis

Názov práce: Simulátor počítačovej hry Gwent

Autor: Patrik Smelík

Katedra / Ústav: Katedra softwaru a výuky informatiky

Vedúci bakalárskej práce: Mgr. Jakub Gemrot, Ph.D.

Abstrakt:

Cieľom tohto projektu bolo vytvoriť simulátor kartovej hry Gwent: The Witcher Card Game v jazyku C#. Simulátor dokáže simulovať hru hráča proti AI a takisto dvoch AI proti sebe. Tým umožní testovať chovanie umelých inteligencií v rôznych situáciách a s rôznymi balíčkami, čo pomôže pri ich vyhodnocovaní a vzájomnom porovnávaní. Zameranie bolo hlavne na jednoduché pridávanie kariet pomocou xml súboru, implementácia umelej inteligencie cez Interface a efektívne načítavanie kariet do programu. Naviac ešte implementujem užívateľské rozhranie pre komunikáciu s používateľom a simuláciu hry v konzole, spolu s editorom pre nastavenie rôznych situácii, ich ukladanie a neskoršie načítavanie.

Kľúčové slová: simulátor, počítačová hra, kartová hra, Gwent, C#

Title: Simulator of the computer game Gwent

Author: Patrik Smelík

Department: Department of software and computer science education

Supervisor: Mgr. Jakub Gemrot, Ph.D.

Abstract:

The goal of this project was to create a simulator for the card game Gwent: The Witcher Card Game in C#. The simulator can simulate games of a player against an AI and also games of two AIs against one another. With that, we can test the behavior of artificial intelligence in several different scenarios and with different card decks, which helps with its evaluation and comparison against other AIs. The main goals were easy card addition using an XML file, the implementation of artificial intelligence through an Interface and effective loading of cards into the program. In addition, the program implements a console user interface for easy player input and a game simulation with several options to create and edit the different scenarios, save them and later load them.

Keywords: simulator, computer game, card game, Gwent, C#

Obsah

1 Úvod

Cieľom tejto práce je vytvoriť simulátor kartovej hry Gwent: The Witcher Card Game (CD Project Red, 2016) ktorý je efektívny, umožňuje jednoduchú definíciu kariet pomocou XML a takisto simuláciu hry hráča proti AI, či dvoch AI proti sebe.

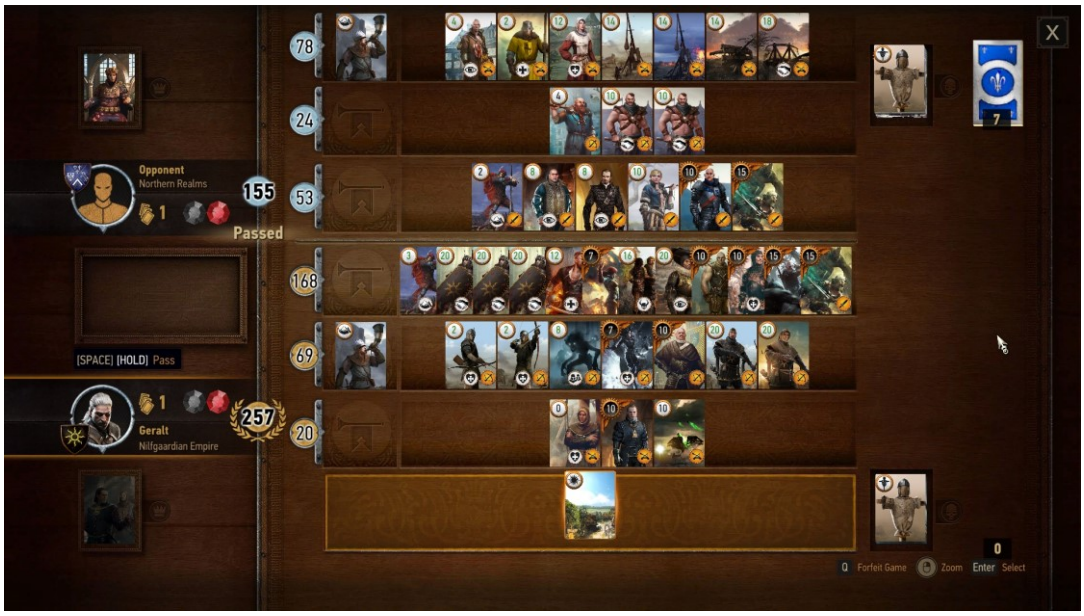
1.1 Gwent

Gwent je kartová hra, ktorá sa prvýkrát objavila ako minihra vo videohre Witcher III: The Wild Hunt (2015) (Obrázok 1). Táto minihra bola natoľko obľúbená, že sa CD Project Red, developer série Witcher (CD Project Red, 2009), rozhodol hru implementovať samostatne, ako multiplayer online „collectible card game“[1](CCG). A tak vznikol Gwent: The Witcher Card Game, ktorý v Máji 2017 vyšiel v Open Beta verzii. (Obrázok 2)

Gwent je hra dvoch hráčov proti sebe. Každý hráč reprezentuje nejakú frakciu, kde si vyberie jedného z možných vodcov a k nemu vytvorí nejaký balík kariet. Hráči sa striedajú vo svojich ťahoch až dokým obaja neskončia, alebo nemajú žiadne karty na zahratie. Vo svojom ťahu hráči pokladajú svoje karty na pole, využívajú ich schopnosti alebo hrajú kúzla. Ale na rozdiel od podobných CCG, ako napríklad Hearthstone (Blizzard Entertainment, 2014), hráč tu nemá žiadny zdroj, pomocou ktorého tieto karty hrá (Hearthstone, mana). Hráči musia počas svojho ťahu zahrať práve jednu kartu, čím vzniká istá komplexita rozhodovania. Ak obaja hráči skončili, body kariet na poli sa zrátajú a kolo vyhrá ten, kto ich má viac. Ten získa bod a začína nové kolo. Hru vyhrá ten hráč, ktorý prvý získa 2 body. Môže tiež nastať aj remíza, ak majú na konci kola obaja hráči rovnako bodov, v tom prípade obaja získajú jeden bod.

Gwent sa takisto odlišuje vo svojom poli. Hra má podľa tvorcov reprezentovať vojnu dvoch vojsk, čiže hráči si nestrážia žiadny svoj život, ako to je napríklad pri Hearthstone-e alebo Magic-u (Wizards of the Coast, 1993). Pole v hre Witcher a neskôr v open bete tvorili 2 strany a na každej 3 rady. Do týchto radov sa hrajú jednotlivé karty. Ale pri vydaní verzie 1.0 boli rady zredukované na 2. (Obrázok 3)

[1] Typ kartovej hry, kde si hráč stavia balík z kariet, ktoré vlastní. Karty získava z “booster” balíkov, ktoré obsahujú X (v našom prípade 5) kariet. Tie balíky sa získavajú buď hraním alebo kupovaním za peniaze.



Obrázok 1: Pôvodný Gwent v hre Witcher III: The Wild Hunt

Obrázok 2: Gwent: The Witcher Card Game v Open Beta verzii (2017)

Obrázok 3: Gwent vo verzii 1.0 (2019)

1.2 Simulátor

Simulátor je program, ktorý sa pokúša simulovať chovanie nejakého reálneho procesu alebo systému, v tomto prípade simuluje chovanie hry, aby sa mohlo skúmať, ako taký systém funguje. Musí obsahovať všetky základné charakteristiky systému, aby sa čo najviac priblížil reálnej veci. Potom sa zmenou niektorých charakteristík môže skúmať jej celkové chovanie.

Využitie takýchto simulátorov je napríklad už spomenuté skúmanie, kde detaily chovania reálneho systému pri zmenách nie sú prístupné, alebo pri spojení s umelou inteligenciou (Artificial Intelligence - AI), kde sa pomocou takého simulátora AI testuje a trénuje. Pri vývoji AI sa často nemôže hneď používať reálna aplikácia, nakoľko to môže byť drahé alebo nebezpečné (AI sa v tejto dobe používa v rôznych odvetviach a použiť netestované AI priamo, napríklad v doprave, by mohlo spôsobiť vážne škody). Preto je ideálne skonštruovať umelé prostredie podobné realite a v ňom umelú inteligenciu testovať, z čoho sa získavajú cenné dáta o chovaní. Následne sa môže AI nejako upraviť, alebo poprípade zmeniť samotné prostredie a skúmať chovanie AI pri takej náhlej zmene, čo by sa v reálnom prostredí takmer určite nedalo.

1.3 Ciele

Cieľom práce je zostrojenie simulátoru, na ktorý kladieme tieto nároky:

=> simulátor by mal implementovať celú hru (pri začatí práce bola aktuálna verzia 1.1)

=> vzhľadom k tomu, že hra Gwent sa stále vyvíja, je potrebná, aby bolo simulátor jednoduché rozšíriť

=> simulátor chceme použiť hlavne na experimenty s umelou inteligenciou, preto musí poskytovať vhodné rozhranie pre implementáciu umelého hráča

=> umelých hráčov budeme chcieť medzi sebou porovnávať, a preto musí simulátor umožniť hru umelých hráčov proti sebe

=> umožniť ladenie konkrétnych inštancií hry

1.4 Related work

V čase písania tejto práce neexistuje žiadny verejne prístupný simulátor Gwentu, preto budem náš simulátor prirovnávať k simulátorom inej kartovej hry, presnejšie k simulátorom hry Hearthstone. Hearthstone je aktuálne asi najznámejšie online CCG, preto k nej existuje aj väčšie množstvo rôznych simulátorov. Spomeniem hlavne simulátory od skupiny HearthSim[2], nakoľko sú zo všetkých asi najznámejšie. Táto skupina udržiava viacero simulátorov s rôznymi prístupmi a implementáciami. Z nich sa zameriam na simulátory SabberStone[3] a Fireplace[4].

SabberStone je simulátor podobný nášmu. Napísaný v jazyku C#, program sa delí na jadro programu kde sú implementované jednotlivé základné schopnosti, filtre podľa ktorých sa vyberajú ciele a tiež jednotlivé karty. Základné atribúty kariet berie simulátor z json súborov, ktoré získava priamo z hry, hocijaké ďalšie schopnosti karty sú poskladané zo základných schopností, ktoré sú implementované v jadre. Ďalšie časti obsahujú testy pre celé jadro, implementáciu užívateľského rozhrania, GUI, a nakoniec rozšírenia, kde je naimplementované nejaké základné AI, konzolová verzia GUI a logovanie.

Náš simulátor je veľmi podobný, takisto využívame filtrovanie pre identifikáciu cieľa schopnosti či tiež máme jadro programu s implementovanými základnými schopnosťami. Najviac sa ale líšime v implementácii kariet. Ako som

[2] https://hearthsim.info/simulators

[3] https://github.com/HearthSim/SabberStone

[4] https://github.com/jleclanche/fireplace

spomínal, SabberStone využíva json súbory priamo z hry. Tam sú reprezentované všetky karty, obsahujúce ID reťazce a ich základné atribúty. Pomocou tohto reťazcu identifikuje karty, zistí atribúty a k danej karte doimplementuje jej schopnosti priamo v kóde. Túto dvojicu uloží a počas hry využíva. Karty sú implementované priamo v kóde, čo je efektívne, ale týmto spôsobom sa ťažko definujú nové karty, hlavne pre ľudí, ktorý s kódom nemajú skúsenosti. Ale využívať definície kariet priamo z hry je určite efektívne, hlavne to zaručuje aktuálnosť kariet. Na druhej strane, hocijaká väčšia zmena môže rozbiť celkové fungovanie karty, čiže taká situácia musí byť strážená.

Fireplace je aktuálne asi najznámejší hearthstone simulátor robený v jazyku Python. Okrem implementácie všetkých základných funkcii využíva podobne ako SabberStone XML súbory obsahujúce definície kariet pre ich základnú implementáciu. Ďalej sa pýši extenzívnymi testami a "built-in" serverom, vďaka ktorému môže komunikovať s oficiálnym hearthstone serverom, alebo simulovať hry cez webové rozhranie.

1.5 Štruktúra textu

Po úvode bude text rozdelený nasledovne:

- Nasledovať bude analytická časť (2). V nej popíšem fungovanie Gwentu. Tiež popíšem architektúru programu, ako funguje jeho jadro, ako som definoval tok hry, ako som definoval karty a aké je užívateľské rozhranie.
- Potom nasleduje programátorská časť (3). Konkrétne popíšem ako som všetko z analytickej časti implementoval a ako sú spolu jednotlivé časti prepojené.
- Nasleduje užívateľská časť (4). Vysvetlím, ako program ovládať a čo jednotlivé príkazy robia.
- Ako ďalšie (5) opíšem, akým spôsobom som sa uistil, že jednotlivé časti programu fungujú ako majú, čiže unit testy.
- A na koniec (6) moju prácu zhrniem, overím všetky svoje stanovené ciele a opíšem, čo som z tejto práce zistil.

2 Analýza

V tejto časti textu sa zameriame na teoretickú prípravu tohto projektu. Vysvetlíme si princíp samotnej hry a ako sa hrá. Hru si následne rozoberieme na menšie časti a preskúmame, čo všetko budeme potrebovať na úspešnú a efektívnu implementáciu.

2.1 Fungovanie hry Gwent

Pri programovaní simulátora sa chceme čo najviac priblížiť k fungovaniu samotnej hry, čiže v prvom rade musíme vedieť, ako hra funguje.

Gwent je hra dvoch hráčov bojujúcich proti sebe. Jedna hra pozostáva z troch kôl a cieľom je vyhrať dve z nich. Hráči sa striedajú v ťahoch, v každom ťahu zahrajú jednu kartu, až dokým jeden nezloží. Tým dáva najavo, že v tomto kole už neplánuje hrať ďalšie karty. Druhý hráč môže pokračovať v hraní kariet dokým chce. Cieľom kola je nazbierať viac bodov ako súper. Pravidlá sa zdajú jednoduché, ale treba myslieť na to, že hru hráč začína s desiatimi kartami a na začiatku druhého kola si potiahne iba dve ďalšie karty, na začiatku tretieho dokonca iba jednu. Ide tu hlavne o spravovanie zdrojov. Gwent je "Collectible card game" (CCG), tým pádom hry sa hrajú s balíčkami, ktoré si hráči poskladajú sami z kariet ktoré vlastnia. Pri skladaní balíka si hráč vyberie frakciu s ktorou chce hrať, karty budú tým pádom musieť patriť tejto frakcii, alebo byť neutrálne. Z takých môžu čerpať všetky frakcie. Balík má vodcu so špeciálnou schopnosťou, plus minimálne 25, ale maximálne 40 kariet. Pri skladaní balíka má hráč nejaký rozpočet (niektorý vodcovia môžu tento rozpočet zvýšiť) a každá karta má nejakú cenu. Hráč tým pádom musí pri stavaní balíka dávať pozor, aby tento rozpočet nepresiahol. Karty sú takisto rozdelené do dvoch farieb, zlaté a bronzové, a v balíku môžu byť najviac 3 rovnaké bronzové karty a iba po jednej z hocijakej zlatej karty.

Hracie karty sú rozdelené do troch hlavných kategórií. "Vojaci" sú karty ktoré sa vyvolávajú na pole, majú vždy nejakú silu a môžu mať aj nejakú jedinečnú schopnosť (Obrázok 4). "Artefakty" sa tiež vyvolávajú na pole, ale nemajú žiadnu silu (vľavo hore na karte je symbol kalicha), majú iba schopnosti, ktoré hráč môže pri svojom ťahu aktivovať. Posledné sú "kúzla", ktoré sa nevyvolávajú (na karte je symbol ohňa), pri zahraní majú iba nejaký efekt.

***Obrázok 4*:** Karta a jej elementy

2.2 Priebeh hry

Keď už vieme ako sa hra hrá, tak sa môžeme pustiť do plánovania. Najprv si vytvoríme stavový automat ťahu jedného hráča (Podrobnejšie v Sekcii 2.5). Automat bude reprezentovať, čo môže alebo už nemôže hráč v jednom ťahu urobiť. Hráč môže počas svojho ťahu buď zahrať jednu kartu, aktivovať kartu, ktorá už leží na poli, alebo ťah skončiť. Hráč musí nejakú kartu zahrať, ak nechce skončiť celé svoje kolo. Takisto, karta sa môže aktivovať predtým aj potom čo sa zahrá karta z ruky. Ak však hráč aktivoval kartu na poli, nemôže potom skončiť svoje kolo a musí zahrať aj kartu z ruky. Kolo hráča sa končí, keď nemá žiadne karty na ruke, čiže už nemá žiadnu kartu, čo môže zahrať, alebo skôr, kedy hráč uzná za vhodné a sám kolo ukončí. Týmto sme si presnejšie určili, ako vyzerá priebeh hry a môžeme začať plánovať jednotlivé časti.

2.3 Hracie pole

Jednou zo základných častí hry je pole, na ktorom sa hra odohráva. Pole sa delí na 4 časti (obrázok 5). Hráčova ruka, odkiaľ hráč hrá karty. Hráčov balík, odkiaľ si karty ťahá. "Graveyard", kam sa presúvajú zničené karty alebo použité kúzla. A nakoniec samotné bojové pole, kam sa vyvolávajú vojaci.

Obrázok 5: Hracie pole

Bojové pole sa delí na dva riadky, "Melee row" a "Range row". Hráč môže kartu zahrať na hociktorý riadok, ale niektoré karty dostávajú špeciálne bonusy, ak sú zahraté na nejaký špecifický riadok. V pôvodnej verzii bol ešte tretí "Siege row", ale od tohto dizajnu sa upustilo kvôli zjednodušeniu hry. Karty boli pôvodne aj obmedzené a mohli byť hrané iba na špecifický row, čo spravilo karty, ktoré mohli byť hrané hocikam veľmi dôležité, ale to bolo tiež nakoniec zmenené. Aj keď sú všetky časti rôzne, slúžia vždy rovnakému účelu, a to skladovaniu kariet v nejakom poradí. Okrem ruky, kde poradie kariet nie je dôležité, ale to nám nevadí. Preto bude najlepšie všetky tieto časti reprezentovať ako zoznamy kariet, z ktorými sa bude počas hry manipulovať a medzi ktorými sa budú jednotlivé karty presúvať.

2.4 Karty

Ďalej potrebujeme preskúmať všetky karty a rozobrať ich na menšie komponenty, ktoré budeme neskôr implementovať. Z tých komponentov budeme potom jednotlivé karty znovu skladať.

V prvom rade si je treba vypísať jednotlivé efekty, ktoré karty môžu aktivovať. Nebudem ich sem vypisovať lebo ich je veľa (viď Appendix A), ale základnými príkladmi sú “Damage”, ktorým sa karte uberie nejaké množstvo sily, alebo “Boost”, ktorý karte naopak silu zvýši. Efekty budú spolu so základnou slučkou hry tvoriť jadro celého simulátora, keďže sa pomocou týchto jednotlivých efektov bude manipulovať so všetkými elementmi hry. Pri skladaní kariet sa budú tieto implementácie využívať na reprezentovanie správneho chovania karty.
Ako som už spomínal, karty sa delia na 3 základné typy, “Vojaci” (Units),”Artefakty” (Artifacts), “Kúzla” (Specials). Ešte existujú špeciálne druhy kúziel, takzvané “Weather”. Sú to efekty, ktoré sa aplikujú nie na karty, ale na celý riadok na poli. Na celý ten riadok potom každé kolo aplikujú svoj efekt, až dokým ich niekto neodstráni, alebo nevypršia. Všetky karty bez ohľadu na typ majú niekoľko spoločných elementov. Každá karta má:

- Názov
- Popis herného efektu (môže byť prázdny)
- Farbu (Zlatá alebo Bronzová)
- Raritu (Common, Rare, Epic, Legendary)
- Cenu (koľko stojí si ju dať do balíka)
- Kategórie (rôzne typy kariet majú rôzne kategórie, nie sú ale povinné)

Takisto sme po zanalyzovaní kariet zistili, že karty s efektami potrebujú niekoľko základných elementov na jeho vykonanie (Príklady nižšie, Obrázok 6). Každý efekt potrebuje nejaký set, čiže miesto odkiaľ bude ciele vyberať, napríklad pole, nepriateľov balík alebo hráčova ruka (Obrázok 6, Avallac’h => From your deck, Geralt => Enemy row). Ďalej potrebuje nejaké upresnenie, čo z daného setu bude vyberať, napríklad iba vojakov, vojaka s najväčšou silou, iba artefakty, nejakú presnú kategóriu alebo dokonca jednu presnú kartu. V poslednom rade ešte potrebujeme počet, čiže koľko z vyfiltrovaných kariet sa vyberie, a tiež spôsob akým sa tie ciele vyberú. Môže to byť náhodne, alebo si z nich vyberie hráč (Avallac’h), či sa efekt uplatní na všetky (Geralt). Efekty si takisto rozdelíme na druhy podľa toho, kedy sa aktivujú. Sú také, ktoré sa aktivujú hneď ako sa karta zahrá, tzv. “Deploy”, efekty ktoré sa aktivujú po tom čo karta zomrie, “Deathwish”, efekty ktoré sú aktivované manuálne hráčom, “Order” a nakoniec pasívne efekty, ktoré sa aktivujú periodicky,

napríklad na začiatku každého ťahu. S týmito všetkými komponentami môžeme poskladať jednotlivé karty.

Príklady (Obrázok 6):
Old Speartip - efekt nemá, je to karta s veľkou silou.
Avallac'h: Sage - Efekt sa aktivuje pri zahratí a iba ak je na Ranged Row. V tom prípade si hráč vyberie zo svojho balíka artefakt a ten zahrá.
Geralt: Igni - Efekt sa aktivuje pri zahratí a iba ak je na Melee Row. V tom prípade karta overí, či má vybratý Row dokopy aspoň 20 sily a ak áno, zničí najsilnejšiu kartu. Ak má viacero kariet najväčšiu silu, zničí ich všetky.

Obrázok 6: Príklady efektov kariet

2.4.1 Reprezentácia kariet

Otázkou je aký spôsob si vyberieme. Karty sa dajú implementovať priamo v kóde, alebo externe, kde sa definujú pomocou textových súborov, alebo súborov XML či JSON. In-code definovanie kariet je efektívne, nakoľko sa karty môžu používať priamo, nevyžaduje to žiadne spracovanie, avšak potom nastávajú problémy s rozšíriteľnosťou. Keďže simulátor sa využíva primárne na testovanie a trénovanie, hodilo by sa mať možnosť karty jednoducho upraviť a potenciálne aj pridať nové. Kvôli tomuto vyberieme externé definovanie, presnejšie pomocou XML, nakoľko to je jazyk, z ktorým som už pracoval a zdá sa mi prehľadný. Ako som ale popísal, tieto definície sa musia nejako spracovávať. Možnosti sú buď pracovať z viacerými súbormi v ktorých budú popísané karty a jednotlivé efekty a informácie získavať vždy z nich. To sa mi ale zdá pomalé, nakoľko neustále otváranie a čítanie textových súborov zaberá veľa času a je to neefektívne. Preto som sa rozhodol mať jeden XML súbor, kde budú popísané všetky karty a z týchto definícii vygenerovať kód samotnej karty. Tie sa potom budú využívať počas hry. Spája to tým pádom rýchlosť "in-code" definície kariet z jednoduchosťou a rozšíriteľnosťou textových definícii. A na

túto generáciu využijeme “template” (šablónu). Template-y sú kombináciou textu a nejakej kontrolnej logiky, pomocou ktorých sa generuje textový výstup. Tu to využijeme na prečítanie definícii kariet z XML súboru a template podľa toho a presnej sady podmienok vygeneruje definíciu karty priamo v jazyku C#. Template knihovní je ale viacero a treba si vybrať. V prvom rade určite narazíte na jednoduché generovacie programy ako XML schema definition tool (“xsd.exe”), ktorý je zabudovaný priamo vo Visual Studiu a vie pretransformovať XML súbor na C# class-u. Problém je že táto generácia je v našom prípade moc jednoduchá a neobsahuje dostatok možností na generovanie celých efektov. Ďalej som narazil na všeobecné textové template-y ako StringTemplate[5], ktoré sa využívajú na generovanie stránok či Java kódu a existuje port aj pre C#. StringTemplate ale pre definovanie logiky využíva svoju vlastnú sadu inštrukcii, kvôli čomu som sa nakoniec rozhodol proti. Lebo neskôr som narazil na Text Template Transformation Toolkit, alebo t4 template už zabudovaný vo Visual Studiu. Len som si k nemu stiahol package “tangible t4”[6], ktorý pri písaní pridáva veci ako intellisense pre zjednodušenie písania. Visual Studio tento template podporuje už dlho a logika sa definuje priamo pomocou jazyka C#. To mi umožnilo preskočiť učenie sa novej syntaxi pre definovanie logiky. Template je takisto jednoduchý na používanie a nevyžadoval takmer žiadne nastavovanie, vďaka čomu som mohol priamo začať.

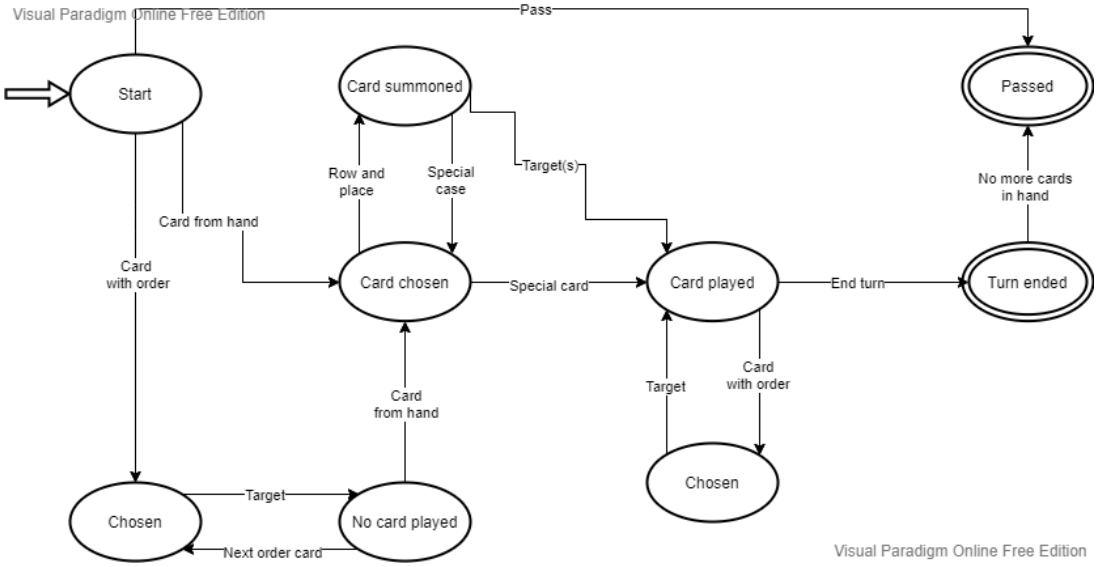
2.5 Hráč

Nasledujúcou dôležitou časťou hry je samotný hráč. Hráč má iba dve komponenty, balík kariet, ktorý si vytvoril a svojho zvoleného lídra. Avšak je potrebnou časťou, keďže hráč určuje, ktoré karty bude počas svojho ťahu hrať, kam ich bude pokladať a aké ciele budú jednotlivé efekty zasahovať. Preto budeme potrebovať nejakú základnú reprezentáciu hráča, z ktorou bude simulátor komunikovať, takzvané API. Jednotlivé implementácie budú potom načítané externe cez dynamické knižnice v prípade umelej inteligencie, alebo implementované priamo v kóde v prípade živého hráča, ktorý bude proti umelej inteligencii hrať. API bude obsahovať iba metódy, ktoré je potrebné implementovať. Z predchádzajúcej analýzy kariet sme zistili, že to budú metódy, ktoré reprezentujú kroky, kde hráč interaguje so simulátorom. Čiže

[5] https://www.stringtemplate.org/

[6] https://marketplace.visualstudio.com/items?itemName=tangibleengineeringGmbH.tangibleT4Editor240plusmodelingtoolsforVS2017

napríklad pri vyberaní rady, kam kartu zahrá, vyberaní cieľa efektu alebo samotnej karty, ktorú chce zahrať. Nič ostatné nebude hráčovi, čiže implementácii, prístupné. Možnosti ťahu som vykreslil v nasledujúcom automate. (obrázok 7).



Obrázok 7: Stavový automat ťahu hráča v hre Gwent

Začína sa samozrejme v uzle *Start*. Hráč má na výber buď nejakú kartu aktivovať (Card with order), vybrať kartu z ruky (Card from hand) alebo svoje kolo skončiť (Pass). Pri aktivovaní karty hráč vyberie nejakú kartu na poli a dostane sa do stavu *Chosen*, následne musí vybrať ciele pre jej efekt (Target), tým sa dostaneme do stavu *No card played*. Odtiaľto môže hráč aktivovať ďalšiu kartu (Next order card) a vrátiť sa o stav späť a proces opakovať, alebo vybrať kartu z ruky (Card from hand). Zo stavu *Card Chosen* vychádzajú 2 cesty. Ak je vybratá karta Vojak alebo Artefakt, hráč musí vybrať Row a miesto, kam chce tú kartu zahrať (Row and place) a dostane sa do stavu *Card summoned*, odkiaľ sa vyberajú ciele efektu (Target), alebo sa odtiaľ v špeciálnom prípade, kde je efekt karty zahratie inej karty, vráti do predošlého stavu (Special case). Ak je karta kúzlo, rovno sa prechádza na výber cieľov (Special Card). Týmto sa dostávame do stavu *Card played*, kde hráč už kartu zahral. V tomto bode môže znovu aktivovať karty na poli (Card with order), čo funguje rovnako ako na začiatku, alebo svoj ťah skončiť (End turn). V stave *Turn ended* sa ťah hráča končí. Ak však hráč už nemá karty na ruke (No more cards in hand) a čiže už nie je možné, aby ďalší ťah nejakú zahral, hráč končí aj svoje kolo. V stave *Passed* hráč ukončil svoje kole a už iba čaká na druhého hráča, dokým aj ten neskončí.

2.6 Testovanie

Nakoniec bude treba otestovať, či to všetko správne funguje. Veci v jadre bude jednoduché otestovať, keďže majú predpísané chovanie. Nám len stačí overiť či sa tie efekty chovajú ako majú. Väčší problém bude testovať chovanie kariet. Keďže každá karta má iné chovanie a sú vždy generované pomocou template-u, nemôžeme pre každú kartu spraviť unikátny test. To by úplne znehodnotilo výhody používania template-u. Avšak môžeme využiť to, že samotné základné efekty sú pevne definované a tiež dobre otestované. A keďže chovanie kariet je potom zložené z týchto efektov, nám teda iba stačí zaručiť, že zahratie hocijakej karty nespôsobí žiadnu výnimku, kvôli čomu by sa vypol program. To otestujeme pomocou nejakých základných scenárov, do ktorých karty zahráme. V tomto nám zase pomôžu template-y, kde si pre každú kartu vygenerujeme jednoduchý test, kde sa karta zahrá do pár základných situácii v hre. A dokým žiadna karta hru nerozbije, máme vyhraté. Takéto testovanie môžeme využiť nie len vďaka template-om, ale aj vďaka ďalšej funkcionalite, ktorý náš simulátor má, a to ukladanie stavu hry.

2.7 Ukladanie stavov

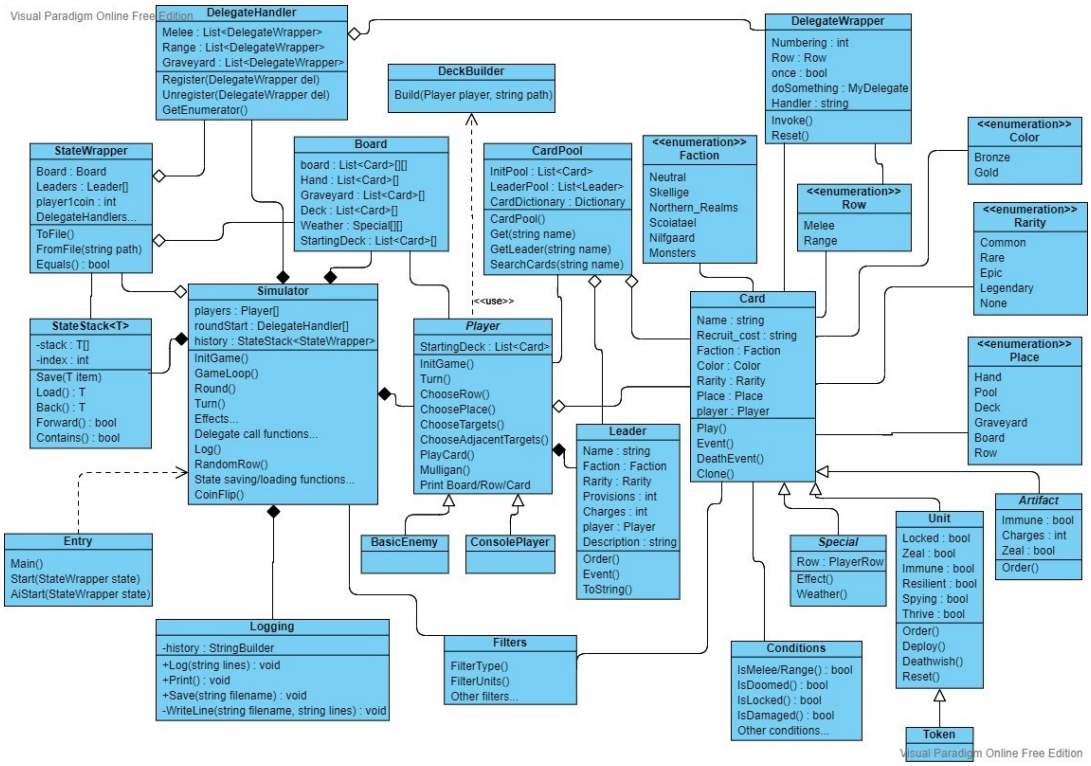
Simulátor vie počas hocijakého bodu v hre uložiť stav hry a pri zapnutí môže tento stav znova načítať. Pod stavom hry myslíme aktuálny stav hracieho poľa plus stav hráča a ktorý z hráčov začal prvý. Toto využijeme tak, že si vopred vytvoríme nejaké základné stavy, napríklad prázdne pole, pár kariet vyložených na poli, plné pole, atď. V každom teste potom postupne načítame jednotlivé stavy a do nich sa pokúsime zahrať danú kartu. A keďže nám ide iba o stabilitu, nič viac nám nebude treba sledovať, čiže testovanie tiež prebehne rýchlo.

3 Implementácia / Programátorská dokumentácia

V nasledujúcej kapitole sa zameriame na samotnú implementáciu jednotlivých častí programu. Postupne budem prechádzať sekcie programu, ktoré sme zanalyzovali v predošlej časti, opíšem ako som ich implementoval a svoje rozhodnutie vysvetlím. Simulátor bol implementovaný v jazyku C# pomocou Visual Studio 2017 a využíva .NET Framework 4.6.1. Na definíciu kariet sa používa XML verzia 1.0 a na lepšiu prácu z t4 template som si stiahol plugin „tangible t4“[7], ktorý pridáva text highlighting a ďalšie pomocné vlastnosti pre jednoduchšie písanie.

3.1 Architektúra

V tejto časti opíšem základnú stavbu programu a ako jednotlivé triedy spolu súvisia. Podrobnejšie ich potom popíšem v ďalších sekciách. Pre vizuálne vykreslenie slúži nasledujúci class diagram (Obrázok 8).



Obrázok 8: UML Class Diagram Simulátoru

[7] https://marketplace.visualstudio.com/items?itemName=tangibleengineeringGmbH.tangibleT4Editor240plusmodelingtoolsforVS2017

Ako si všimneme, náš simulátor má 3 hlavné triedy, `Simulator`, `Player` a `Card`, okolo ktorých alebo s ktorými pracujú všetky ostatné. `Simulator` je trieda, ktorá sa stará o chod hry, je to takzvané jadro. Vytvára a stará sa o reprezentáciu hracieho poľa v triede `Board`, volá delegáty uložené v triede `DelegateHandler` a zapisuje všetko čo sa deje pomocou triedy `Logging`. Takisto ukladá stav hry do triedy `StateWrapper`, a jednotlivé stavy ukladá do štruktúry `StateStack`. Samotná trieda `Simulator` a aj hráči sú vytvorený v triede `Entry`, čo, ako názov hovorí, slúži ako vstup do programu. Trieda `Player` je abstraktná trieda, v ktorej sú implementované nejaké základné metódy a z ktorej dedia konkrétne implementácie hráča, či umelého alebo reálneho. Základný umelý hráč je už implementovaný v triede `BasicEnemy` a reálny hráč je zas implementovaný v triede `ConsolePlayer`. Hráč má vždy nejakého lídra, ktorý je potomok triedy `Leader`, a balík kariet postavený pomocou statickej triedy `DeckBuilder`. Balík je pole obsahujúce karty a tie sú získavané z triedy `CardPool`, ktorá na začiatku inicializuje všetky jednotlivé karty a uloží ich pre neskoršie klonovanie, využíva teda návrhový vzor Prototype. Základné informácie karty sú definované v triede `Card`. Niektoré základné informácie bývajú vyberané z predpísaných hodnôt v takzvaných enumeráciách. Sú to presnejšie frakcia z enumerácie `Faction`, rarita z `Rarity`, farba karty z `Color`, miesto karty z `Place`, či riadok karty z `Row`. Karty sú potom rozdelené do rôznych typov a ich definície sú v triedach `Special`, `Unit` a `Artifact`, ktoré od triedy `Card` dedia. Nakoniec samotné karty dedia od týchto jednotlivých typov. Ešte jedným špecifickým typom vojaka je trieda `Token`, ktorá má presne definovaný stav a vytvára sa iba počas hry. Karty väčšinou implementujú efekty, ktoré sú využívané počas hry. Na ich implementáciu karta využíva metódy z triedy `Simulator`. Ďalej na upresnenie typu cieľa využíva statickej triedy `Filters` a na definíciu podmienok, ktoré musia byť splnené, využíva triedu `Conditions`. Nakoniec existujú efekty, ktoré sa neaktivujú okamžite, ale sú aktivované periodicky vždy, keď sa niečo stane. Také efekty sú uložené ako delegáty a zabalené do triedy `DelegateWrapper`. Ten sa potom cez triedu `Simulator` zaregistruje a uloží sa do správnej inštancie triedy `DelegateHandler` podľa typu.

3.2 Pole

V prvom rade potrebujeme implementovať hracie pole (Sekcia 2.3, *Board.cs*). Je to základný krok, v podstate to reprezentuje aktuálny stav hry (okrem stavu hráča) a všetky ostatné časti programu budú s týmto poľom nejak pracovať, buď z neho budú čítať informácie, alebo nejakým spôsobom meniť jeho stav. Ako som spomínal v analýze, Pole sa rozdeľuje na 4 časti: Samotné hracie pole, Ruka, Balík a Graveyard. Všetky ale plnia rovnakú úlohu, a to uskladňovať karty v určitom poradí. Preto som sa ich rozhodol všetky implementovať ako `List`-y (Ukážka kódu 1). S tými sa jednoducho pracuje a implementujú rozhranie `IEnumerable`, keby to chcela nejaká umelá inteligencia externe reprezentovať inak. A keďže je hracie pole symetrické, každý hráč má vždy svoju ruku, nejaký balík, Graveyard a svoju časť hracieho poľa, môžeme všetky tieto `List`-y potom jednoducho držať v poliach veľkosti 2. Na začiatku hry sa potom iba každému hráčovi priradí index, pomocou ktorého budeme k týmto `List`-om pristupovať. Index 0 sa priradí hráčovi, čo hru začínal (vyhral hod mincou), a index 1 hráčovi druhému. Jedinou výnimkou budú Rows, keďže tie sú u každého hráča dve. Tam budeme preto potrebovať pole polí. Postupovať sa ale bude obdobne, prvým indexom sa určí hráč a druhým riadok (Melee=0 alebo Range=1). Takisto do tejto časti pridáme separátne pole kariet pre Weather efekty. Keďže tie sa aplikujú na riadok, ale nie sú v skutočnosti súčasťou toho riadku (nezaberajú na ňom miesto), budeme ich mať uložené separátne. Pristupovať sa k nim bude rovnakým spôsobom ako ku samotnému riadku, nevráti sa ale `List` reprezentujúci riadok, ale iba samotná karta, poprípade null ak žiadna nie je. Ešte jedna extra časť bude `Starting Deck`, alebo balíček s ktorým hráč začínal, keďže niektoré karty vyberajú z neho. Avšak počas hry sa nijako nemení, čiže nám stačí si na začiatku hry okopírovať balíky oboch hráčov a tie si uložiť ako `readonly List`-y.

```
public class Board
{
    internal List<Card>[] Hand = { new(), new() };
    internal List<Card>[] Graveyard = { new(), new() };
    internal List<Card>[] Deck = { new(), new() };
    internal List<Card>[][] board = { new[] { new List<Card>(), new
List<Card>() }, new[] { new List<Card>(), new List<Card>() } };
    internal Special[][] Weather = { new Special[2], new Special[2] };
    internal readonly List<Card>[] StartingDeck = { new(), new() };
}
```

Ukážka kódu 1: *Board.cs*

3.3 Jadro

Pokračovať budeme ústrednou časťou celého programu, jadrom (*Simulator.cs*). Ako som spomínal v predošlej Sekcii 3.2, že pole reprezentuje stav hry, tak táto časť, jadro, reprezentuje jej funkcionalitu. Implementuje simuláciu spolu s rozhraním hráča a stará sa o správny chod simulátora. V Sekcii 2.1 som opísal pravidlá hry. To nám poslúži na načrtnutie jej priebehu. Na začiatku hry potrebujeme inicializovať hráčov. Okrem konfigurácie hry, ktorou sa budem zaoberať neskôr musím rozhodnúť, ktorý hráč začína. Tým sa mu aj priradí číslo, ktorým ho budeme po celú dobu reprezentovať. Tiež si uložíme ich začiatočné balíky (viď predošlú sekciu) a hráčom uložíme, kto je ich nepriateľ. Hra pozostáva z maximálne troch kôl (dokým jeden hráč nevyhrá dve). Počas jedného kola sa hráči striedajú v ťahoch až dokým obaja neskončia. Začína hráč ktorý vyhral hod mincou alebo predošlé kolo. Počas ťahu môže hráč aktivovať karty, ktoré má na poli a buď zahrať jednu kartu z ruky alebo kolo skončiť. Hráč kolo skončí aj ak už nemá kartu na zahratie (viď stavový automat obrázok 7). Hlavný loop hry teda pozostáva z kôl až dokým jeden hráč nemá dve výhry (Ukážka kódu 2). Medzi kolami sa prečistí hracie pole a hráči si potiahnu pár kariet. Kolo potom spočíva v striedaní ťahov medzi hráčmi až dokým obaja neskončia (Ukážka kódu 3). Vyhráva ten, kto má viac bodov, čo zistíme z hracieho poľa. Skončenie kola jednoducho reprezentujeme boolean-om, zatiaľ čo počet výhier má hráč uložené ako číslo. Hráč má tiež na začiatku hry, ako aj medzi kolami možnosť vrátiť až 3 karty späť do balíka a potiahnuť si nové. To je implementované v metóde `Mulligan`.

```
private void GameLoop()
{
	var first = 0;
	while (player1.State.Wins != 2 && player2.State.Wins != 2)
	{
		player1.State.Passed = false;
		player2.State.Passed = false;
		Log("Round start.");
		//Ak toto nie je prvé kolo, každý hráč ťahá 3 karty
		if (player1.State.Wins != 0 || player2.State.Wins != 0)
		{
			for (var i = 0; i <= 1; i++)
			{
				Draw(players[i], 3);
				var hand = GetHand(players[i]);
				foreach (var card in players[i].Mulligan(hand, 3))
				{
					ShuffleInto(GetDeck(players[i]), card);
					card.Place = Place.Deck;
					Draw(players[i], 1);
				}
			}
		}
		Round(first);
		//Na konci kola sa kontroluje, kto vyhral
		if (GetBoard(player1).Strength() >= GetBoard(player2).Strength())
		{
			player1.State.Wins++;
			first = player1.State.Coin;
			Log(player1.Name + " wins this round.");
		}
```

Ukážka kódu 2: *Simulator.cs, hlavný game loop hry*

```
private void Round(int first)
{
	OnRoundStart();
	while (!(player2.State.Passed && player1.State.Passed))
	{
		for (var i = first; i < 2; i++)
		{
			currentPlayer = players[i].Name;
			player1Destroyed = false;
			player2Destroyed = false;
			Log("Turn start " + currentPlayer);
			Turn(players[i]);
			if (pcOnly && Console.KeyAvailable)
			{
				Console.ReadKey();
				AiMenu(players[i]);
			}
			Log("Turn end " + players[i].Name);
		}
		first = 0;
	}
	OnRoundEnd();
}
```

Ukážka kódu 3: *Simulator.cs, Priebeh jedného kola*

Okrem toku samotnej hry jadro ovláda aj manipuláciu so všetkými jej časťami. Preto si implementujeme aj všetky možné efekty, ktoré budú karty potom volať. Tie efekty budú potom manipulovať s hracím polom, kartami alebo hráčmi podľa potreby. V Sekcia 2.4 som spomínal, že ich je veľa (takisto viď Appendix A), ale aspoň načrtnem spôsob, akým to bude prebiehať. Napríklad efekt, ktorý som zmienil v analytickej časti, Damage, má jednoduchú implementáciu. Metóda `Damage` zníži silu karty o nejaký počet a ak ten počet dosiahne nulu, karta sa zničí (Ukážka kódu 4).

```
public void Damage(int amount, IEnumerable<Card> targets)
{
      foreach (var card in targets.FilterUnits().Cast<Unit>().ToList())
      {
            Log(card.Name + " was damaged by " + amount);
            card.Strength -= amount;
            OnCardDamaged(card);
            if (card.Strength <= 0)
            {
                  Destroy(card);
             }
      }
}
```

Ukážka kódu 4: *Simulator.cs, efekt Damage*

Na druhej strane, metóda `Destroy` (Ukážka kódu 5), ktorá kartu ničí potrebuje už trochu komplikovanejšiu implementáciu. Najprv sa musíme pozrieť, či karta nie je "Doomed". Také karty totiž neputujú do Graveyardu, ale sú kompletne odstránené z hry. Tým sa ničenie takej karty končí. Ak ale nie je Doomed, zavoláme "Deathwish" efekt karty. Deathwish efekt sa spustí, keď je karta zničená. Následne môžu existovať karty, ktorých efekt sa aktivuje, keď sa niečo stane, napríklad ak je nejaka karta zničená. Také efekty máme registrované ako delegáty a uložené v `List`-och podľa podmienok a vždy keď tá situácia nastane sa všetky delegáty aktivujú (podrobnejšie v Podsekcii 3.4.1, kde je aj popis všetkých situácií, na ktoré karty môžu reagovať). Po ich aktivácii sa karta musí presunúť z hracieho poľa do Graveyard-u. Takisto, ak to bola karta z nejakým aktívnym delegátom, ten delegát musí byť odhlásený. Tiež existujú karty, ktorých delegáty sa naopak registrujú, keď sú presunuté do Graveyard-u, čiže následne treba zavolať túto registráciu. Pri presunutí sa tiež karta resetuje do pôvodného stavu. Nakoniec sa na hráčovi, ktorého karta bola zničená zapíše, že sa taká vec stala kvôli interakciám ďalších kariet. A týmto je konečne karta riadne zničená.

```
public void Destroy(Unit card)
{
      if (card.Doomed)
      {
            SetDestroyedFlag(card.player);
            Banish(ref card);
            return;
      }
      Log(card.Name + " was destroyed.");
      card.Deathwish(this);
      OnCardDestroyed(card);
      GetRow(card.player, card.Row).Remove(card);
      var handler = GetHandler(card.EventDelegateWrapper?.Handler);

      handler?[card.player.State.Coin].Unregister(card.EventDelegateWrapper);
      card.Place = Place.Graveyard;
      GetGraveyard(card.player).Add(card);
      card.DeathEvent(this);
      card.Reset();
      SetDestroyedFlag(card.player);
}
```

Ukážka kódu 5: *Simulator.cs, efekt zničenia karty*

Takto postupne naimplementujeme každý jeden efekt v hre s tým, že sa budeme držať reálneho chovania v originálnej hre. Nesmieme preto zabudnúť na ani jednu možnú interakciu, presne ako v metóde `Destroy`.

Nasledujú metódy na volanie už spomínaných delegátov (viď Podsekciu 3.4.1) a zopár pomocných metód ako `CoinFlip`, ktorá hodí mincou alebo `RandomRow`, ktorá náhodne určí riadok. Takisto ešte metóda `Log` ktorá zapisuje jednotlivé volané metódy, pre jednoduchšie debugovanie alebo keď si hráč chce pozrieť doterajšiu históriu hry.

Nakoniec sa jadro stará o uchovávanie stavov, ich ukladanie do súboru a tiež ich následné načítanie. Aktuálny stav sa skladá z hracieho poľa, stavu hráča a informácii o tom, kto začínal. Pre uloženie stavu som následne vytvoril classu *StateWrapper.cs*, do ktorej sa všetky informácie zapíšu v metóde Save (Ukážka kódu 6). Jadro si po uložení stavu tento Wrapper zapíše do Stack-u (*StateStack.cs*), odkiaľ ho na pokyn hráča môže znovu načítať. Na začiatku si môže hráč uchovávať maximálne 10 stavov, avšak toto číslo sa môže meniť. Myslím, že 10 stavov nie je pamäťovo náročné a pritom je to dostatočne veľké číslo pre potreby testovania a ladenia AI. Hráč má možnosť ísť v hre o krok späť alebo krok dopredu, avšak hráč si sám určuje kedy si stav hry uloží. Pri kroku späť sa načíta posledný stav a následne sa ukazovateľ stacku posunie. Stack jednotlivé stavy nevymazáva aby sa hráč následne mohol vrátiť do pôvodného stavu, z ktorého začal, keby nič v minulých stavoch

nechcel zmeniť. Stavy sa ale premažú, ak hráč niekoľkých krokoch späť začne normálne hrať. Hráč môže takisto stav uložiť do súboru, alebo nejaký stav zo súboru počas hry načítať.

```
private StateWrapper Save()
{
      return new()
      {
            Player1Coin = player1.State.Coin,
            Board =
            {
                  board = new[]
                  {
                        new[] {new List<Card>(board.board[0][0]), new
                        List<Card>(board.board[0][1])},
                        new[] {new List<Card>(board.board[1][0]), new
                        List<Card>(board.board[1][1])}
                  },
                  Hand = new[] {new List<Card>(board.Hand[0]), new
                  List<Card>(board.Hand[1])},
                  Graveyard = new[] {new List<Card>(board.Graveyard[0]), new
                  List<Card>(board.Graveyard[1])},
                  Deck = new[] {new List<Card>(board.Deck[0]), new
                  List<Card>(board.Deck[1])},
                  Weather = new[] {board.Weather[0], board.Weather[1]}
            },
            RoundStart = roundStart,
            RoundEnd = roundEnd,
            TurnEnd = turnEnd,
            TurnStart = turnStart,
            CardPlayed = cardPlayed,
            CardDamaged = cardDamaged,
            CardDestroyed = cardDestroyed,
            CardDiscarded = cardDiscarded,
            Leaders = new[] { players[0].Leader, players[1].Leader }
      };
}
```

Ukážka kódu 6: *Simulator.cs, zabalenie stavu do wrapper-u*

3.4 Karty

Karty sú hlavným zameraním tejto hry. Hraním kariet hráči menia stav hry, ovplyvňujú hracie pole a ultimátne vďaka nim vyhrávajú. S kartami teda bude pracovať takmer každý element simulátora. V Sekcii 2.4 som spomínal, že všetky karty bez ohľadu na typ majú niekoľko spoločný základných elementov ako názov, popis, atď. (Ukážka kódu 7). Vďaka tomu si môžeme spraviť jedného spoločného predka, ktorý bude obsahovať tieto základné informácie, plus pár základných metód, ktoré sa môžu používať pre všetky typy. Táto classa (*Card.cs*) bude potom obsahovať aj metódy označené ako *virtual*, ktoré si potom jednotlivé karty môžu prepísať podľa potreby.

```
public class Card
{
      public DelegateWrapper EventDelegateWrapper;

      public string Name { get; protected set; }
      public int Recruit_cost { get; protected set; }
      public string Categories { get; protected set; }
      public Faction Faction { get; protected set; }
      public Color Color { get; protected set; }
      public Rarity Rarity { get; protected set; }
      public Place Place { get; set; }
      public bool Doomed { get; set; }
      public Player player { get; set; }
      public bool Visible = true;
      internal int Base_Damage { get; set; }
      internal int Damage { get; set; }
      internal int Base_Boost { get; set; }
      internal int Boost { get; set; }
      internal bool available { get; set; }
      protected string Description { get; set; }

      public virtual void Play(Simulator simulator)
      {
```

Ukážka kódu 7: *Card.cs, základné informácie karty*

S týmto spoločným predkom môžu potom pracovať všetky metódy, ktoré nepotrebujú viac ako tie základné informácie, napríklad pri presúvaní kariet metóde stačí vedieť iba hráča a kde sa aktuálne karta nachádza. V prípade metódy zameranej len na niektoré typy, ako metóda `Summon`, ktorá nebude na hracie pole vyvolávať Kúzla, si metóda skontroluje, akého typu je predaná karta a zachová sa podľa výsledku.

Jednotlivé typy budú samozrejme dediť od základnej class-y `Card`, a ku nej si pridajú svoje špecifické elementy.

Vojaci (*Unit.cs*) napríklad majú vždy nejakú silu alebo môžu mať určené Flagy ako "Doomed" (viď Appendix C), ktorá po zničený kartu odstráni z hry namiesto presunutia do Graveyard-u. Takisto bude obsahovať metódy `Order`, `Deathwish` a `Deploy`, ktorá sa budú volať podľa toho, kedy sa efekt danej karty aktivuje (Deathwish pri zničení karty, Deploy pri zahratí a Order pri manuálnom aktivovaní). Tieto metódy nebudú mať telo a budú označené ako *virtual*. Existujú, aby sa dali zavolať bez toho, aby simulátor poznal názov samotnej karty a označenie *virtual* umožňuje samotným kartám ich prepísať, aby v nich mohli definovať samotný efekt. Tiež ich nemôžeme označiť ako *abstract* (keby sme classu definovali ako abstraktnú), lebo nie všetky karty majú tieto efekty, niektoré dokonca nemajú žiadny efekt. Dalo by sa to ešte urobiť tak, že ich označíme ako abstraktné a karty bez

daného efektu ju implementujú ako prázdnu, ale to ich môžeme rovno definovať ako prázdne tu a karta si override-ne metódy, ktoré potrebuje.

Artefakty (*Artifact.cs*) sú podobné, avšak nebudú obsahovať až toľko informácii alebo metód. Ich efekt sa aktivujú iba manuálne, alebo periodicky ako delegát. Takisto nemajú silu alebo nemôžu mať toľko rôznych Flag-ov.

Kúzla (*Special.cs*) zas nemajú žiadne extra informácie, jedine efekty aplikované na Row, Weather, si budú pamätať na akom Row sa nachádzajú. Takisto nemajú viac typov aktivovania efektov, keďže to sú kúzla. Budú mať iba metódu `Effect`, ktorý sa aktivuje okamžite, a `Weather`, ktorý sa registruje ako delegát a bude aktivovaný periodicky.

Samotné karty budú nakoniec dediť od tohto svojho typu. Vysvetlené v Podsekcii 2.4.1, karty budú generované pomocou template-u. Template bude poznať cestu k XML súboru, z neho bude brať definície jednotlivých kariet a podľa sady pravidiel vygeneruje potrebný kód. Vygenerované karty s efektami budú využívať implementované základné efekty v jadre (popísané v minulej Sekcii 3.3), plus doplnkové metódy na filtrovanie či kontrolovanie podmienok, ktoré karty musia splniť predtým než sa efekt aktivuje.

3.4.1 Delegáty

Efekty kariet sa normálne aktivujú raz (Deploy pri zahratí, Deathwish pri zničení a Order manuálne), ale sú niektoré karty, ktorých efekty sa aktivujú opakovane, vždy keď niečo nastane. Takéto efekty je treba periodicky volať. Normálne by sa dalo tieto efekty uložiť ako `event` so špecifickou podmienkou aktivácie, ale Gwent potrebuje všetky takéto karty aktivovať v poradí Melee Row, Range Row, postupne zľava doprava, zatiaľ čo event by ich spúšťal v poradí, ako boli karty zahraté. Preto tieto efekty uložíme ako delegát a samotný delegát uložíme do triedy `DelegateWrapper`. Táto trieda bude okrem delegáta obsahovať ďalšie špecifikácie spustenia (či bude efekt spustený raz, maximálne x krát alebo až po y ťahoch), poradie delegáta podľa polohy karty, a takisto typ podmienky, podľa ktorej bude efekt spúšťaný. Môže to byť presnejšie buď na začiatku alebo na konci ťahu, na začiatku alebo na konci kola, ak je zahratá nejaká karta, ak je nejaká karta discardnutá (odhodená z ruky do Graveyard-u) alebo ak je nejaká karta zničená. Toto zabalenie sa deje už pri vytvorení karty. Po zahratí karty sa tento wrapper musí

registrovať ako aktívny. Na to má trieda `Simulator` vytvorených niekoľko štruktúr typu `DelegateHandler`, presne podľa typu aktivácie. Špecifickým prípadom je kúzlo typu Weather, ktoré sa registruje do samotného poľa na `Board`-e a nie do handlera. Táto trieda obsahuje polia delegátov, do ktorých ich ukladá v správnom poradí. Trieda tiež implementuje rozhranie `IEnumerable` pre jednoduché enumeráciu delegátov. Následne sme vytvorili pre každý handler metódy, ktoré postupne aktivujú všetky ich delegáty. Nakoniec už iba stačí, aby sa tieto metódy zavolali v správny moment, čo jednoducho zakomponujeme do metód (Destroy, Discard) alebo na správne miesto v game loop-e (pred začiatkom ťahu/kola a po jeho konci). Po odstránení karty z poľa sa delegát odregistruje.

Na filtrovanie slúži statická classa *Filters.cs*, kde máme naimplementované statické metódy, ktoré filtrujú listy podľa rôznych kritérií (viď Appendix B). Množiny kariet, či už to sú ciele efektov alebo zoznam kariet na aplikovanie nejakých zmien, sa predávajú medzi metódami pomocou zoznamov, teda presnejšie pomocou rozhrania `IEnumerable`. Filtrovaním teda v tomto prípade myslíme nejakú úpravu týchto zoznamov (Ukážka kódu 8). Filtre sú rôznorodé, či už podľa typov kariet, podľa kategórii, alebo keď už je vyfiltrovaný určitý typ, tak môžeme filtrovať aj presnejšie podľa sily.

```
/// <summary>Filter out all cards of a specific <paramref name="name"/>.
/// </summary>
/// <param name="input">List of cards from which to filter.</param>
/// <param name="name">Name of the card to filter out.</param>
public static IEnumerable<Card> FilterCard(this IEnumerable<Card> input,
string name)
{
      return from c in input where c.Name == name select c;
}
/// <summary>Filters out all damaged units.</summary>
/// <param name="input">List of cards from which to filter.</param>
public static IEnumerable<Card> FilterDamaged(this IEnumerable<Card> input)
{
      return from unit in input.Cast<Unit>() where unit.Strength <
      unit.Base_Strength select unit;
}
```

Ukážka kódu 8: *Filters.cs, príklady filtrov, `FilterCard` vráti všetky karty daného mena, `FilterDamaged` vráti všetky karty, ktoré sú Damaged, čiže majú silu menšiu ako pôvodnú*

Statická classa *Conditions.cs* zas obsahuje rôzne podmienky. Tie sa využívajú, keď sa karte efekt aktivuje iba za určitých podmienok, alebo sa po ich splnení aktivuje nejaký extra efekt (Ukážka kódu 9). Podmienky sú implementované ako extension metódy, čiže sa môžu volať priamo na nejakú kartu alebo ich zoznam. Uľahčuje a sprehľadňuje to kód pri jeho generovaní. Podmienky sú takisto rôzne. Môže sa kontrolovať hráč, či vlastník karty má niečo na ruke alebo na poli, alebo priamo karta, či je nejakého typu, kategórie, alebo či má aspoň nejakú silu.

```
public static bool ControllingFaction(this Player player, Simulator simulator,
Faction faction, int amount = 1)
{
      return simulator.GetBoard(player).FilterFaction(faction).Count() >=
      amount;
}

public static bool ContainsCategory(Player player, Simulator simulator, Place
place, string category, int amount = 1)
{
      var where = simulator.GetPlace(place, player);
      return where.FilterCategory(category).Count() >= amount;
}
```

Ukážka kódu 9: *Conditions.cs, príklady podmienok,* `ControllingFaction` *zistí, či má hráč na poli aspoň jednu kartu danej frakcie,* `ContainsCategory` *zistí, či je na danom miesta nejaká karta určenej kategórie*

Obrázok 9: Príklady kariet

Pre konkrétny príklad, karta "Iorveth's gambit" (Obrázok 9, kód v triede *Iorveths_Gambit.cs*) zahrá 2 náhodné karty kategórie "Trap" z balíka, ak sme mali na začiatku hry v balíku aspoň 4 takéto karty. Do efektu sa to preloží tak, že karta najprv zavolá metódu `ContainsCategory` z triedy *Conditions.cs* ktorá kontroluje, či je na danom mieste aspoň istý počet kariet danej kategórie. V našom prípade bude miesto "Starting deck", kategória "Trap" a počet 4. Ak sa táto podmienka splní, karta vezme hráčov štartovací balík a pomocou metód z *Filters.cs*

odtiaľ vyfiltruje Artefakty, z nich karty kategórie “Trap” a z nich náhodne vyberie dve. Tieto karty nakoniec zahrá.
Karta “Gabor Zigrin” (kód v triede *Gabor_Zigrin.cs*) zasa pri zahratí skontroluje, na ktorý Row bol položený a podľa toho bude mať priradený token (Resilience alebo Immunity). Následne si aktivuje pasívny efekt, ktorý sa zavolá vždy pri zahratí karty. Efekt skontroluje, či je zahratá karta kategórie “Dwarf” a ak áno, tak sa Gabor boostne o 1. Na skontrolovanie kategórie samozrejme slúži metóda `IsCategory`.

3.4.2 XML

Najprv vysvetlím definovanie kariet v XML (opisovanie Sekcie 2.4 v jazyku XML). Karta bude reprezentovaná v elemente **Card** a ako atribúty bude mať jej základné informácie, čiže jej typ, farbu, raritu. V elemente Card bude definované jej meno, kategórie, popis, frakcia ku ktorej patrí, jej cena a sila, a ak to jej efekt vykonáva, hodnotu Damage alebo Boost. Nasledovať bude element **Tokens**, do ktorého sa definujú všetky Flagy, ktoré daná karta má. Môže byť samozrejme aj prázdny. Posledný je element **Ability**, kde sa definuje efekt danej karty. Karta môže mať aj viaceré Ability elementy, avšak každý musí byť iného typu, inak by sa prekrývali. Ak má jeden typ viacero efektov, to sa dá definovať vo vnútri elementu. Typ bude definovaný ako atribút. Element bude ďalej obsahovať element **Set**, ktorým sa určí, odkiaľ sa budú brať ciele karty. Miesto elementu Set môže byť element **Target**, ktorým sa určí presné meno karty, na ktorú sa to bude aplikovať. Na tomto mieste sa to väčšinou používa, keď je cieľom karty ona sama, píšeme meno “this”. Set bude obsahovať element **Selection**, čo určí, kto bude vyberať tie ciele (hráč) alebo či budú vybraté náhodne, poprípade či budú vybraté rovno všetky. Okrem neho bude obsahovať element **Filter**, ktorý určí, aký typ z daného setu bude vyberať. Filter má ako atribúty ešte rôzne možnosti na upresnenie cieľa, ako napríklad presnú kategóriu. Namiesto filtra môže byť takisto použitý spomínaný element Target, keď z daného setu afektujeme všetky kópie jednej presnej karty. Filter bude v sebe obsahovať elementy **Count**, ktorým sa určí počet afektovaných kariet a element **Effect**, ktorým sa určuje efekt, čo sa na dané vybraté karty aplikuje. Efekt je určený v jeho atribúte a všetky sú definované v DTD súbore (v prílohe „Gwent Simulator.zip“ priečinok Cards). Niektoré efekty, ako napríklad Transform, budú potrebovať extra element na upresnenie. Transform vyžaduje v elemente Effect element Target, aby sa vedelo, na akú kartu sa daný cieľ má premeniť. Alebo efekt Draw, vyžaduje element **Amount**,

na určenie, koľko kariet sa má ťahať. Namiesto elementu Effect môže byť ešte element **Condition**, ktorým sa skontroluje, či pre daný Set/pre daný Target platí nejaká podmienka. Do tohto elementu sa potom znovu určuje Set/Target, pre ktorý sa nakoniec určí aj efekt. Je to tak urobené preto, lebo existujú karty, ktoré na jednej množine skontrolujú požiadavku, a ak platí, na inej množine aplikujú efekt. Condition môže byť takisto definovaný v elemente Effect pre prípad, že sa po prvom efekte skontroluje podmienka, napríklad či bola nejaká karta zničená, a po jej splnení sa má stať nejaký iný efekt. Pri takýchto vnorených efektoch sa v elemente Target môže namiesto mena použiť kľúčové slovo "previous", ktorým sa určí posledný cieľ, na ktorý bol aplikovaný nejaký efekt.

Set môže byť ešte zabalený v elemente **Event**, ak má daná karta pasívny efekt (napríklad Weather alebo Order). Všetko sa určuje normálne, ale Event bude okrem Set/Target obsahovať aj element **Frequency**, kde sa pomocou atribútov určí kedy a za akých podmienok sa tento efekt aktivuje.

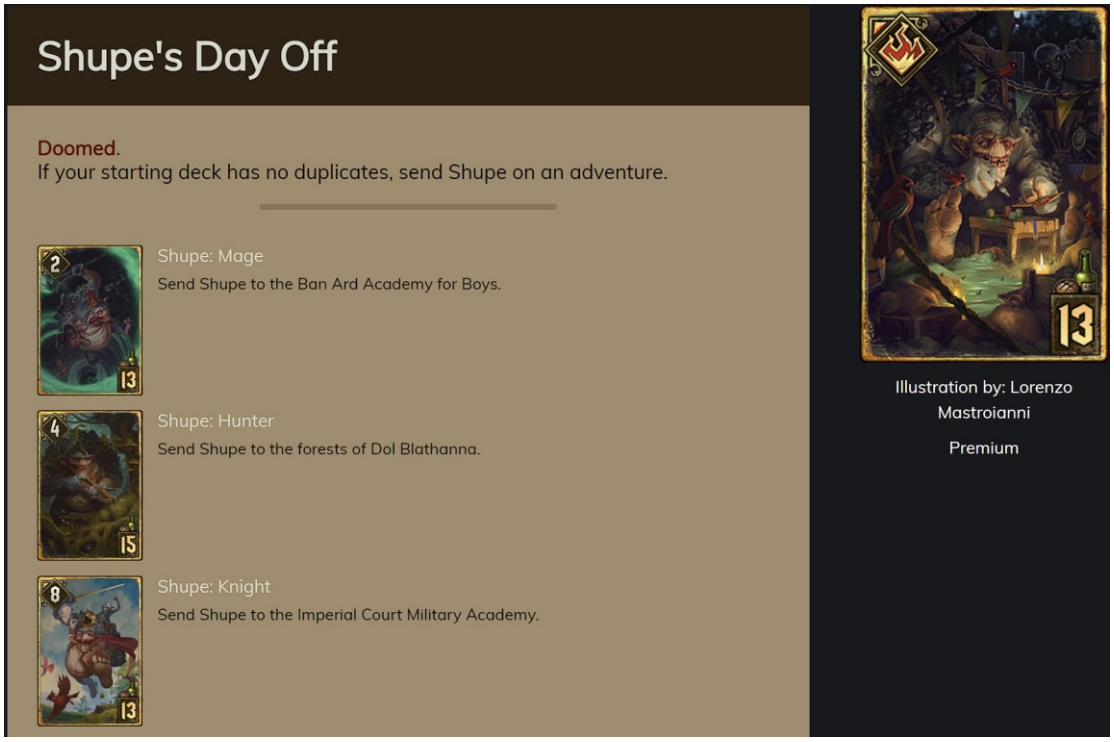
Element Ability môže takisto obsahovať element **Place**. Ten zas určuje, že sa efekt stane, len keď je karta na nejakom presnom riadku. Pokračuje sa rovnako, ako už bolo opísané.

```
<Card Color="Gold" Rarity="Legendary" Type="Unit">
    <Name>Caretaker</Name>
    <Categories>Relict</Categories>
    <Strength>3</Strength>
    <Recruit_Cost>12</Recruit_Cost>
    <Faction Name="Neutral"/>
    <Description>
     Deploy (Ranged): Play an artifact from your graveyard.
    </Description>
    <Ability Type="Deploy">
      <Place Row="Ranged">
        <Set From="Graveyard" Whose="Player">
          <Selection Type="Player"/>
          <Filter Type="Artifact">
            <Count>1</Count>
            <Effect Type="Play"/>
          </Filter>
        </Set>
      </Place>
    </Ability>
  </Card>
```

Ukážka kódu 10: *Cards.xml, popis karty v XML*

Avšak bohužiaľ, nie všetky karty je možné reprezentovať v tejto xml stavbe. Napríklad karta "Shupe's day off" (Obrázok 10) má na výber 3 rôzne verzie karty, z ktorých každá dá ešte na výber z 3 rôznych efektov, ktoré sa neoplatí po jednom

vypísať a uložiť do nášho XML súboru. Preto som takúto funkcionalitu, ktorú má iba pár špecifických kariet, neimplementoval a template ich spracovanie nepodporuje. Takisto by to bolo veľmi komplikované na vypísanie v tejto XML štruktúre.



Obrázok 10: Karta „Shupe's day off"

3.4.3 Template

Template na generovanie kódu sa delí na tri súbory. V prvom súbore (*CardTemplate.tt*) sa určí cesta ku správnemu XML súboru. Template ďalej tento súboru otvorí a kartu po karte bude posielať do druhého súboru, kde sa bude podľa spísaných pravidiel generovať kód. Po vygenerovaní kódu sa zavolá metóda z tretieho súboru (*MultipleOutput.tt*), ktorá vygenerovaný kód uloží. Tretí súbor je stiahnutý[8] a vďaka nemu máme umožnené pomocou jedného template-u vytvárať viacero súborov. Normálne má totiž jeden template jeden výstup. Najdôležitejšou časťou je druhý súbor (*XmlElementsHandler.tt*), kde sú spísané všetky pravidlá. Generovanie funguje na princípe spracovávania jednotlivých elementov v samostatných metódach a komunikáciou týchto metód medzi sebou. Pre každý element existuje metóda ktorá ho spracuje a každá metóda vie, ktoré ďalšie elementy

[8] https://web.archive.org/web/20160501025241/http://www.olegsych.com/2008/03/how-to-generate-multiple-outputs-from-single-t4-template

sa v danom elemente môžu nachádzať. Metóda teda po spracovaní skontroluje existenciu pod-elementov, ktoré pošle na spracovanie. Kvôli takému zanáraniu si template neustále pamätá, v akej hĺbke bola aktuálna metóda volaná. Metódy potom toto číslo využívajú na pomenovanie premenných, aby sa názvy medzi sebou nebili a tiež aby boli premenné v tej istej hĺbke správne využité. Na začiatku sa vypíšu potrebné importy pre classu a pomocou konštruktora sa zaplnia základné informácie karty. Ďalej sa doplnia informácie podľa typu karty. Potom sa načítajú "Ability" elementy a začne sa ich spracovanie. Samotné spracovanie potom prebieha presne podľa stavby XML súboru. Je ale potrebné brať do úvahy každú možnú kombináciu elementov a správne naimplementovať jednotlivé rozdiely. Výroba template-u trvala zďaleka najdlhšie, ale ako som opisoval pri vyberaní reprezentácie kariet, markantne nám to ušetrí čas, keďže nám stačí vyrobiť jeden univerzálny template, ktorý sa bude používať na všetky karty. Takisto je aj vcelku jednoduché ho rozšíriť. Novo naimplementovaný efekt sa iba pridá do `HandleEffect` metódy, ktorá spracováva jednotlivé efekty, ako ďalšia switch vetva. Poprípade pre pridanie nových atribútov karty sa vytvorí nová handle metóda, ktorá ho bude spracovávať. Samotný atribút potom stačí zakomponovať do XML stavby, čiže definovať v DTD so správnymi obmedzeniami a pridať, v ktorých elementoch sa môže nachádzať.

3.5 Hráč

Nakoľko cieľom simulátora je poskytovať rozhranie pre implementovanie umelého hráča, potrebujeme hráča implementovať tak, aby sa od neho dalo dediť. A keďže do samotného hráča (*Player.cs*) implementujeme aj nejakú základnú funkcionalitu, označíme classu ako abstract (Ukážka kódu 11). Abstraktné classy, na rozdiel od klasického interface-u môžu obsahovať aj plnohodnotné metódy. Metódy, ktoré budú musieť byť implementované externe označíme tiež ako abstraktné a tým si zaručíme, že všetko potrebné bude fungovať.

```
public abstract class Player
{
	public PlayerState State;
	private int _coin;
	protected Simulator simulator;
	public Player opponent;
	public PlayerRow MeleeRow;
	public PlayerRow RangeRow;
	public Leader Leader { get; private set; }
	public List<Card> StartingDeck = new();
	public string Name;
	protected bool FirstPrint;
	protected bool Ordered;
	protected const int OptionsStart = 3;
	protected const int OptionsEnd = 16;
	private const int CardWidth = 7;

	/// <summary>
	/// Sets players leader to <paramref name="newLeader"/>.
	/// </summary>
	/// <param name="newLeader"></param>
	public void SetLeader(Leader newLeader)
	{
		Leader ??= newLeader;
		Leader.player = this;
	}
```

Ukážka kódu 11: *Player.cs, základné parametre hráča*

Classa *Player.cs* obsahuje hráčov stav (*PlayerState.cs*), kde sa ukladajú hráčove základné informácie, ako počet výhier, líder, alebo aká strana mince mu bola priradená. Hráč ďalej obsahuje svoj začiatočný balík a referencie na nepriateľa, svoje Rows na poli a na samotný simulátor pre jednoduché odkazovanie sa a volanie efektov. Základné metódy, ktoré sme potrebovali implementovať sú buď inicializačného charakteru alebo vykreslovacie pre potreby konzolového rozhrania. Inicializačné sú metódy ako `SetLeader` ktorý uloží lídra a `InitGame`, ktorý vytvorí nový stav. Inicializácia balíka potom nastáva v konstruktore, kde classa dostane cestu k textovému súboru a takisto meno hráča pre potreby logovania. V konstruktore sa následne zavolá funkcia `Build` z DeckBuilder-a. Tú budeme ďalej rozoberať v Sekcii 3.6.

Vykreslovacie metódy sú metódy `PrintCard`, `PrintRow` a `PrintBoard`, ktoré vykreslia do konzoly kartu, riadok a celé pole v tomto poradí. Takisto tu sú implementácie pre editovacie menu a výber cieľa. Bližšie si ich rozoberieme v Sekcii 3.7.

Nakoniec classa obsahuje abstraktné metódy, ktoré budú implementované buď ako reálny hráč s možnosťami interakcie pre užívateľa, alebo externe cez DLL knihovnu.

Toto označenie majú všetky metódy, kde sa vyžaduje nejaký input od hráča. Metódy ako `Mulligan`, kde si hráč vyberá, ktoré karty vráti späť do balíka, `ChooseTargets`, kde si má hráč vybrať ciele pre daný efekt, či `Turn`, kde má byť implementovaný samotný ťah hráča.
Okrem tejto triedy obsahuje simulátor aj základnú implementáciu umelého hráča (*BasicEnemy.cs*). Je vytvorená hlavne kvôli testovacím potrebám a pre každú implementáciu iba jednoducho vyberie náhodnú podmnožinu zoznamu, ktorý mu je predaný a ten vráti.
Takisto obsahuje aj implementáciu reálneho hráča (*ConsolePlayer.cs*), kde sa pre všetky potrebné implementácie metód na konzolu vypíše menu, z ktorým bude užívateľ interagovať. Implementácia menu čaká na input z klávesnice a podľa neho sa chová. Väčšinou je input šípka, pre pohyb v menu, a Enter alebo ESC pre potvrdenie či zrušenie výberu. Vypisovanie kariet či poľa funguje rovnako ako pri ostatných menu.

3.6 Príprava hry

Po vysvetlení jednotlivých častí simulátora sa môžeme zaoberať jeho spustením. Je niekoľko vecí, ktoré treba urobiť predtým, ako sa hra začne. Program sa začína v súbore *Entry.cs*. Pri spustení máme na výber dva módy, hráč vs AI a AI vs AI. Hru môžeme spustiť novú, alebo načítať nejakú, ktorú sme si predtým uložili. Môžeme ju takisto spustiť v móde “NoVisual”, ktorý hru spustí bez grafického rozhranie a iba po jej dokončení vypíše výsledok. Tento mód je samozrejme dostupný len pre hru dvoch umenlých hráčov.
Po nastavení hry musíme obom hráčom priradiť nejaký balík a pre umelú inteligenciu máme možnosť načítať externú knižnicu, ktorá bude implementovať nášho hráča. Ak mu knižnicu neurčíme, načíta sa základná implementácia, ktorá bude ťahy robiť náhodne. Po tomto všetkom sa vytvorí inštancia simulátora a začne sa príprava. V prvom rade si inicializujeme všetky karty, ktoré máme naimplementované a uložíme si ich do `Dictionary` pre jednoduchý prístup. Na toto slúži statická classa *CardPool.cs* , ktorá vo svojom statickom konstruktore pomocou reflekcie získa všetkých potomkov classy `Card` a po jednom ich inicializuje (Ukážka kódu 12). Potom si túto inštanciu a jej meno uloží ako dvojicu do slovníka. Takto môžeme potom jednoducho pristupovať ku hocijakej karte. Toto isté sa urobí pre všetkých lídrov, nakoľko tí sú separátny od kariet. Trieda ešte

obsahuje aj metódy, ktoré vrátia inštancie karty a lídrov, plus metódu `Search`, ktorá prehľadáva slovník pre názvy začínajúce sa na daný `string` a nakoniec vráti ich zoznam.

```
static CardPool()
{
      var types = Assembly.GetAssembly(typeof(CardPool)).GetTypes();
      var init = from cardType in types
                 where cardType.IsSubclassOf(typeof(Card)) && cardType !=
                 typeof(Unit) && cardType != typeof(Artifact) && cardType !=
                 typeof(Special) && cardType != typeof(Leader) && cardType !=
                 typeof(Token) select cardType;
      var leaders =from cardType in types
                   where cardType.IsSubclassOf(typeof(Leader)) && cardType !=
                   typeof(Leader) select cardType;
      foreach (var cardType in init)
      {
            var card = Activator.CreateInstance(cardType) as Card;
            InitPool.Add(card);
            CardDictionary.Add(card?.Name ??
                       throw new InvalidOperationException("This card type does
                       not exist: " + cardType), card);
      }

      Pool = InitPool.AsReadOnly();

      foreach (var leaderType in leaders)
      {
            var leader = Activator.CreateInstance(leaderType) as Leader;
            LeaderPool.Add(leader);
      }
}
```

Ukážka kódu 12: *CardPool.cs, inicializovanie kariet pomocou reflexie a ich ukladanie do slovníka v statickom konštruktore*

Toto sa využíva pri ovládaní a editovaní hry. Viac to budeme rozoberať v ďalšej sekcii.

Ďalej je potrebné vytvoriť balíky pre oboch hráčov. To sa deje v súbore *DeckBuilder.cs*. Je to takisto statická classa a volá sa pri vytváraní nového hráča. Balíček sa vytvára rozparsovaním textového súboru, ktorý mu bol predaný. V ňom by mal byť správne definovaný balík. Okrem získavania kariet z Poolu, DeckBuilder takisto kontroluje, či je taký balík možný. To znamená, či všetky karty patria pod správnu frakciu, či ich je správny počet a tiež či nebol presiahnutý cenový strop, ktorý daný líder mal. Ak hocijaký z týchto pravidiel nie je platný, vyhodí sa výnimka.

Po tomto všetkom sa už len správne inicializujú hráči a pole, poprípade sa načíta stav a hra sa môže začať.

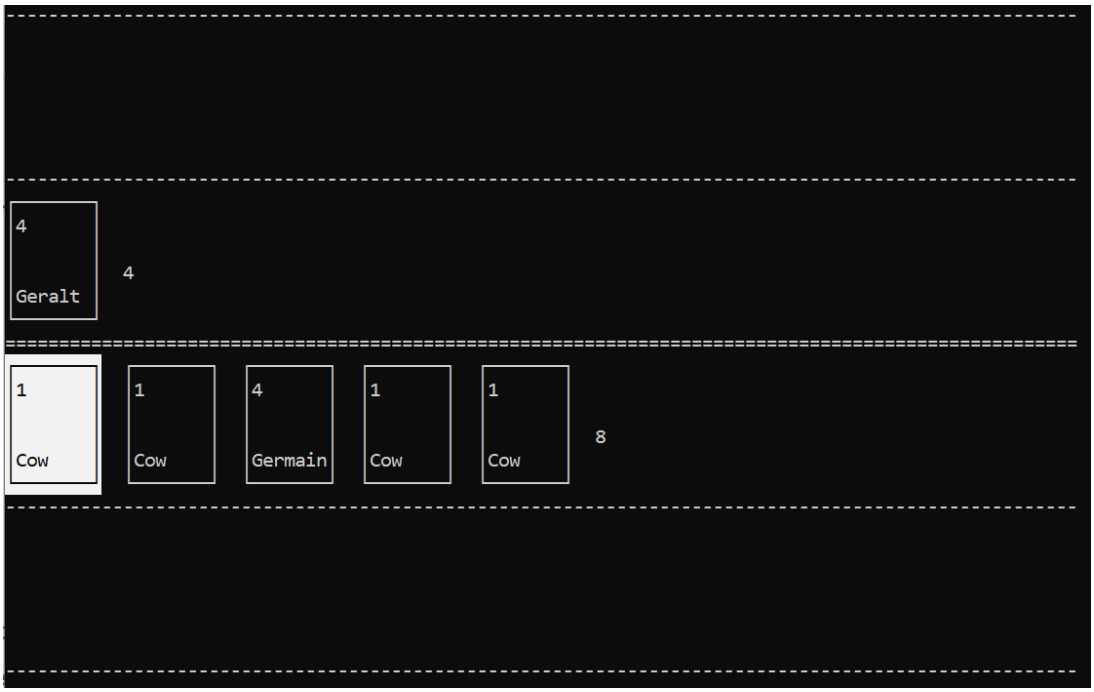
3.7 Konzolové rozhranie

Poslednou dôležitou časťou je konzolové rozhranie, pomocou čoho sa celý simulátor bude ovládať. Tento simulátor je naimplementovaný ako konzolová aplikácia, čiže veľa, čo sa týka konkrétnych výstupov, opisovať nebudem. Popíšem ale všetky časti, kde sa bude čakať nejaký input od užívateľa.

Ako som opisoval v minulej Sekcii 3.6, po spustení programu sú na výber dva módy hry, Hráč vs AI, a AI vs AI. Takisto je možné začať hru novú, alebo hru načítať z uloženého stavu. Možné je aj spustiť hru dvoch AI proti sebe v móde bez vykreslovania ťahov, čiže hra bude bežať na pozadí a na konci sa zobrazí výsledok, plus možnosť si pozrieť históriu ťahov. Po určení balíkov hráčom a poprípade načítaní externej knižnice sa hra začne. Opíšem najprv hru hráča proti AI, keďže väčšina interakcí prebieha práva v tomto móde, a potom opíšem, akým spôsobom sa v rámci rozhrania odlišuje hra dvoch umelých hráčov.

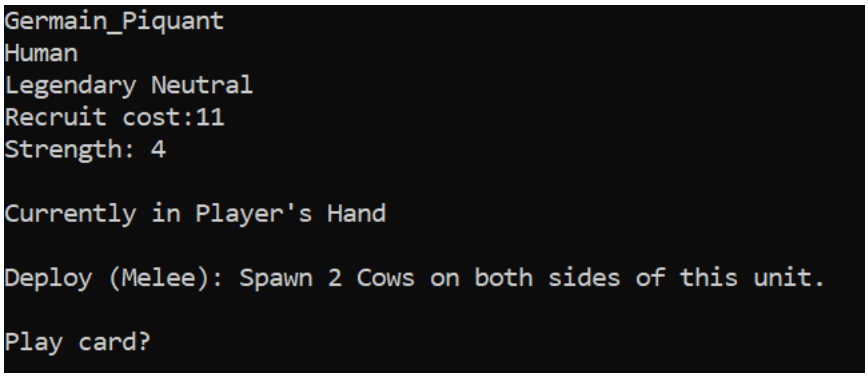
Po začatí hry je užívateľovi vykreslená jeho začiatočná ruka a má možnosť vrátiť až tri karty do balíka, na miesto nich si potiahnuť nové. Program beží v konzole a ovláda sa klávesnicou. Karty sú vykreslené jednoducho pomocou ASCII znakov, ale zobral som si príklad z jedného fóra na stránke stackexchange[9] a karty vykreslil ako obdĺžniky so silou alebo znakom v ľavom hornom rohu a názvom v dolných dvoch riadkoch (Obrázok 11).

[9] https://codereview.stackexchange.com/questions/82103/ascii-fication-of-playing-cards



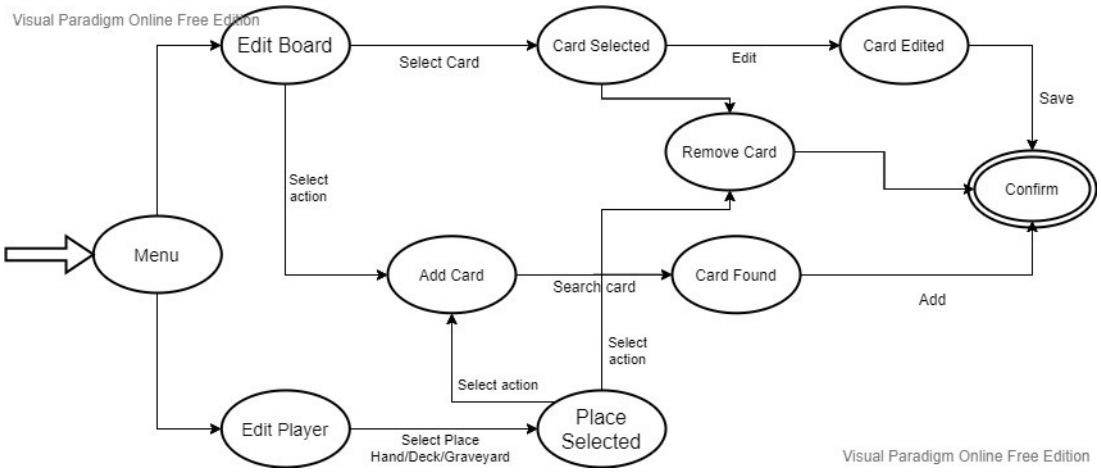
Obrázok 11: Vykreslenie poľa aj s kartami v Simulátore

Po vybratí kariet sa začne hra. Pole som vykreslil ako 4 riadky s dostatkom miesta na vykreslenie kariet. Na pravo v každom riadku je vypísaná jeho celková hodnota. Počas nepriateľovho ťahu hráč vidí pole pred nepriateľovím ťahom, po čom sa vypíše, aký ťah nastal a ako pole vyzerá po ňom. Počas svojho ťahu má hráč k dispozícii menu s možnosťami, čo môže urobiť.
Menu obsahuje zobrazenie jednotlivých častí hry, ako svojej ruky odkiaľ môže hrať karty, bojového poľa, odkiaľ sa môže pozrieť na karty, ktoré tam ležia a poprípade svoje aktivovať, atď. Menu takisto obsahuje aj kontrolné možnosti ako možnosť editovať stav hry, uloženie hry, zobrazenie histórie ťahov alebo ukončenie hry.
Zobrazenie poľa som už opísal, zobrazenie ruky, či balíka má formu vypísania zoznamu kariet, odkiaľ si môžete nejakú vybrať. Po vybratí sa zobrazia jej bližšie informácie (Obrázok 12) a odtiaľ je možnosť sa vrátiť späť, alebo tento výber potvrdiť, ak to bol zoznam ruky hráča.



Obrázok 12: Detail karty po vybratí z ruky.

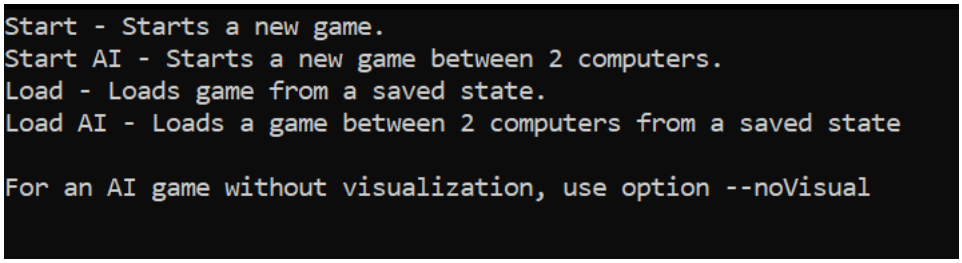
Vypísané informácie sú určené podľa typu karty a implementované v `ToString` metóde jednotlivých typov. Týmto spôsobom sa vyberá karta na zahratie z ruky. Zoznam sa vygeneruje aj po zahratí karty, ak daná karta má nejaký efekt a hráč potrebuje vybrať cieľ. Možnosť History vypíše históriu ťahov hry. Počas hry sú všetky dôležité udalosti logované, táto možnosť ich iba vypíše na obrazovku. Ostatné možnosti budem opisovať v užívateľskej dokumentácii, ich funkcionalitu som popísal v predošlých sekciách.
Jednou možnosťou, ktorou sa ale ešte budeme zaoberať je Edit (Obrázok 13). To prepne simulátor do editovacie stavu a dá užívateľovi možnosť manipulovať so stavom hry. Na výber dostanete editovať pole alebo jedného z hráčov. Možnosti editácie sú odobratie, pridanie karty alebo manipulácia informácii na karte. Odobratie karty proste kartu z daného miesta odstráni. Pridanie karty spustí kvázi "search engine", kde máte zadať názov karty a táto funkcia vám vráti zoznam, z ktorého si môžete vybrať. Nakoľko implementovaných kariet nie je veľa, funkcia vygeneruje nový zoznam po pridaní alebo odobratí hocijakého znaku. Pri väčšom počte kariet bude potom efektívnejšie začať prehľadávať až od napríklad tretieho pridaného znaku. Zoznam kariet sa generuje z CardPool-u a slúži na to metóda `Search`, ktorú som spomínal v predošlej sekcii. Po vybratí karty sa karta z Poolu naklonuje a pridá na správne miesto. Pri pridávaní na pole je potom ešte treba určiť, kam daná karta pôjde, inak tento spôsob funguje všade rovnako. Pri editovaní kariet sa zobrazia bližšie informácie karty, avšak s pár bude možné interagovať. Veci ako sila alebo počet Charge-ov bude možné upraviť, alebo sa dá pridať či odstrániť tokeny ako "Doomed" alebo "Locked". Tieto zmeny sa zapamätajú, avšak uložia sa až po potvrdení týchto zmien kliknutím na možnosť save. Po tomto sa zmeny uložia na samotnú kartu.



Obrázok 13: Stavový diagram editovania.

4 Užívateľská dokumentácia

Simulátor je postavený ako spustiteľný EXE súbor, ale takisto sa dá otvoriť solution vo Visual Studiu a spustiť tam. Simulátor je konzolová aplikácia, čiže po spustení sa otvorí konzola a cez ňu sa interaguje s aplikáciou. Konzola berie vstup z klávesnice, prevažne sa bude treba hýbať v menu pomocou šípiek. Výber z menu sa potvrdí klávesou Enter, či zruší klávesou Esc. V ojedinelých prípadoch, ako napríklad po spustení alebo pri editovaní kariet simulátor očakáva napísaný názov alebo príkaz.



Obrázok 14: Začiatočná konzola po spustení Simulátora.

Po spustení simulátora sa otvorí konzola z možnými módmi spustenia. Po zadaní konfigurácie sa spustí hra.

Pri móde AI vs AI užívateľ neinteraguje s hrou. Každý ťah je vykreslený na konzolu, ak nebol zvolený mód noVisual. Avšak pri stlačení klávesy sa otvorí editačné menu, kde má užívateľ možnosť si pozrieť históriu ťahov a editovať aktuálny stav hry, či sa v histórii stavov posunúť dopredu alebo dozadu. Takisto ma k dispozícii zmeniť konfiguráciu hry.

História ťahov sa zobrazí ako text, kde každý riadok obsahuje jednu akciu. Posledná akcia je zobrazená v prvom riadku a smerom dole sú postupne vypísané staršie akcie.

Pri zmene konfigurácie hry má užívateľ možnosť zapnúť už spomínaní mód “noVisual”, zmenšiť delay, čo zrýchli vykresľovanie ťahov, alebo zapnúť takzvaný “stepMode”, ktorý namiesto automatického vykresľovania bude po každom ťahu vyžadovať stlačenie hocijakej klávesy. To umožňuje krokovanie a lepšiu analýzu hry.

Pri zvolení editovania stavu hry si užívateľ najprv vyberie, ktorú časť hry chce editovať. Má na výber hracie pole alebo jedného z hráčov. Editovanie funguje vždy rovnako, po vybratí miesta je na výber z toho miesta kartu odobrať, alebo naopak tam kartu pridať. Pri odoberaní sa jednoducho zvolí karta a tá sa z toho miesta odoberie. Pri pridávaní sa zobrazí vyhľadávacie pole, kam treba napísať názov karty, ktorá sa chce pridať. Pri písaní sa takisto bude zobrazovať zoznam kariet

začínajúcich na zadané znaky, čiže nie je treba písať celý názov. Po nájdení a zvolení karty sa karta pridá na dané miesto. Pri editovaní hracieho poľa je takisto možné editovať kartu ktoré sa už nachádzajú na poli. Po zvolení karty sa otvorí menu, kde je možné zmeniť silu karty, veľkosť Damage-u alebo Boost-u, či aktivovať karte tokeny ako "Doomed" či "Immune". Zmeny nakoniec treba uložiť zvolením možnosti "Save".

Tieto možnosti sú takisto dostupné aj pri móde hráča vs AI. V tomto móde, ale užívateľ priamo interaguje s hrou, nakoľko je hráčom. Hráč má pri svojom ťahu okrem už spomínaných funkcií možnosť si pozrieť stavy jednotlivých častí hry. Z ruky má možnosť zahrať nejakú kartu a pri prezeraní poľa má možnosť aktivovať nejakú so svojich zahratých kariet. Program sa inak chová ako normálna hry s tým rozdielom, že sa ovláda pomocou klávesnice. Takisto má možnosť uložiť stav hry, či načítať stav z uloženého textového súboru.

Pre vytvorenie nových kariet si treba prečítať Podsekciu 3.4.2, kde opisujem stavbu XML súboru. Po pridaní novej karty do súboru *Cards.xml* si je treba byť istý, že je napísaná správne. Na to slúži priložený súbor *Cards.dtd*, ktorý by mal automaticky kontrolovať XML súbor a pri nejakej nesprávnej interakcií danú chybu vypísať. Po kontrole správnosti je treba spustiť template. Ten následne vygeneruje kód karty. Template sa spúšťa cez súbor *CardTempalte.tt*, stačí ho otvoriť vo visual studiu stlačiť Ctrl+S, alebo pri otvorenom projekte kliknúť na súbor pravým tlačidlom a zvoliť "Run custom tool".

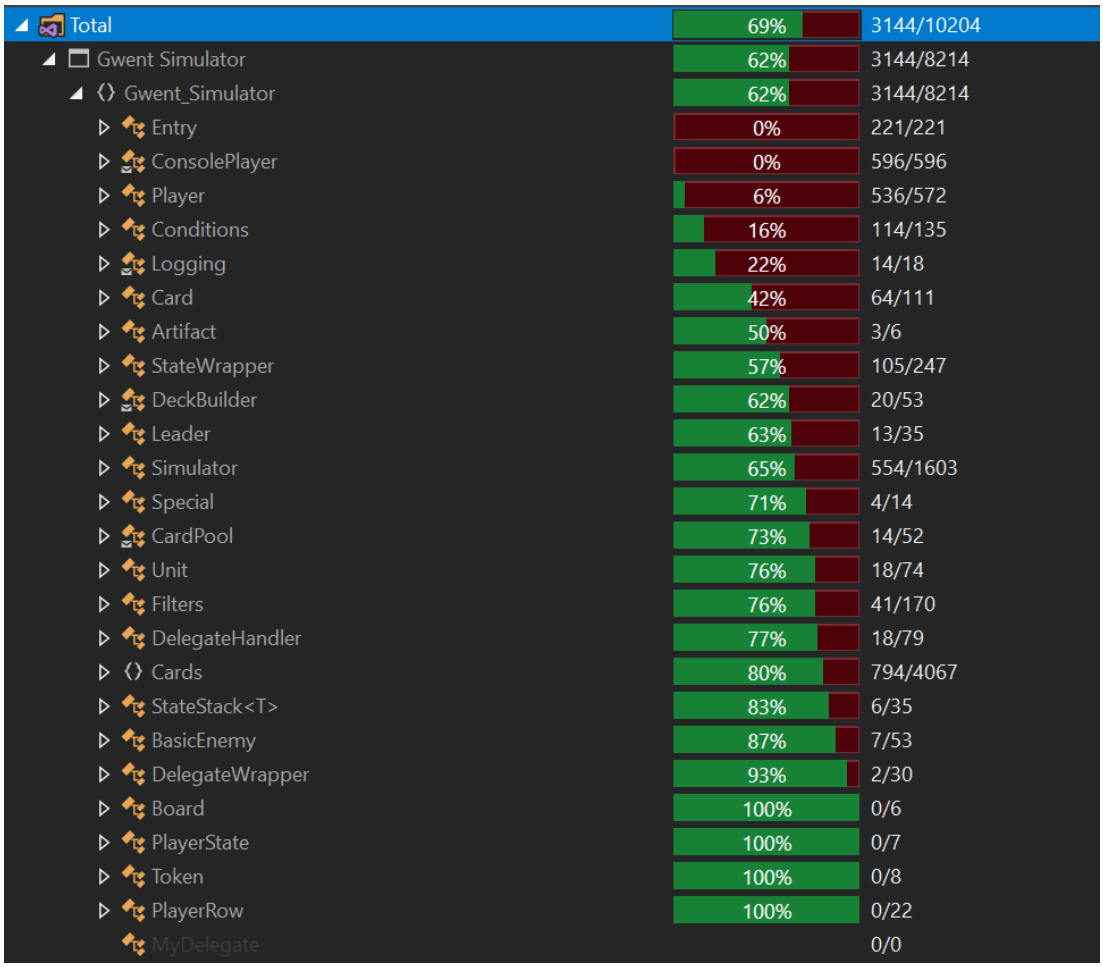
Nakoniec pre vytvorenie umelej inteligencie, je treba vytvoriť dynamicky linkovanú knižnicu (súbor .DLL), ktorá bude obsahovať implementáciu danej umelej inteligencie. Pri jej vytváraní si je treba pridať môj projekt ako referenciu. Classa, v ktorej sa bude AI implementovať následne musí dediť od classy `Player`. Treba potom naimplementovať všetky potrebné metódy. Metódy sú označené ako abstraktné, čiže pri zlej imlementácii to bude vyhadzovať chybu. Pri vytváraní DLL cez visual studio musí byť "Target framework" nastavený na ".NET standard". Okrem týchto požiadaviek nie je žiadne iné obmedzenie na implementáciu. Po zapnutí programu potom stačí zvoliť skompilovaný DLL súbor ako AI implementáciu a program ju začne využívať.

5 Evaluácia / Testovanie

Po implementácii nášho program je treba otestovať, či všetko funguje, ako má. Ako som spomínal v Sekcii 2.6, pri takomto programe sme veľa vecí, ako vykresľovanie, načítanie stavov, logging či inicializácia a fungovanie hráčov testovali ručne. Vieme, že tieto veci fungujú, vďaka tomu, že vieme program spustiť, hra funguje ako má a že všetko sa vypisuje korektne. Preto sa hlavne zameriame na testovanie funkčnosti efektov a kariet. Tiež som spomínal, že efekty majú predpísané chovanie a nám stačí overiť, či robia to, čo majú. Preto pre každý efekt vytvoríme vhodný stav hry, tam aplikujeme efekt a následne skontrolujem, či sa stalo to, čo sa malo stať. Napríklad, efekt "Boost" zvyšuje silu karty o dané číslo. Na otestovanie si vytvoríme kartu a uložíme hodnotu jej sily. Následne na kartu aplikujeme efekt. Potom už len skontrolujeme, či má karta správne zvýšenú silu. Takto vytvoríme testy pre všetky efekty implementované v jadre a máme o základ postarané.

Ešte by sa nám hodilo otestovať jednotlivé karty, ale ako bolo spomenuté, karty sú veľmi rôznorodé a písať testy ručne pre všetky karty nám vynuluje výhodu, ktorú máme generovaním kariet cez template. A tu môžeme znovu využiť generovanie. Vytvoríme niekoľko stavov, do ktorých sa karty budú hrať. Stavy budú rôznorodé, pre otestovanie čo najväčšieho počtu možností. Vybral som preto 4 rôzne hustoty zaplnenia poľa, a to pole prázdne, a postupne 1,3 a 5 kariet na každom riadku. Pre každú kartu potom vygenerujeme test, kde sa postupne tieto stavy načítajú a karta sa pri danom stave zahrá,aktivuje a následne zničí. Takto otestujeme všetky možné aktivácie v rôznych situáciách. A keď sa nič nepokazí, máme vyhraté. Týmto sme pomocou generovania otestovali všetky karty a to aj dosť efektívne. Tiež som k testom pridal jedno spustenie hry dvoch umelých hráčov proti sebe, čo potvrdí, že všetky kontrolné metódy ako inicializácia, hlavný loop hry, atď. funguje ako má.

Celkové pokrytie projektu týmito unit testami je 69% (obrázok 15), ale sústredíme sa iba na dôležité časti ako jadro a karty, čiže to sme očakávali. Triedy, ktoré nie sú testami úplne pokryté sú hlavne triedy týkajúce sa vykreslovania do konzole, presnejšie trieda `Player`, či metóda `ToString` v triede `Card` a jej potomkoch. Ďalej triedy vyžadujúce vstup od hráča ako trieda `ConsolePlayer` či metódy rôznych druhov menu v triede `Simulator`. Tiež metódy ukladania stavov v triede `Simulator` či ukladania histórie do súboru v triede `Logging`. Všetky tieto časti boli testované ručne počas vývoja a debugovania.



Obrázok 15: Pokrytie kódu testami.

6 Záver

Prácu sme začali s cieľom vytvoriť simulátor hry Gwent, ktorý bude možné používať na testovanie umelých inteligencii. Takisto sme chceli jednoducho reprezentovať karty pre možnosť vytvorenia nových kariet, a následne ich efektívne spracovávať. Tieto požiadavky mali nakoniec byť zabalené v prehľadnom užívateľskom rozhraní, cez ktorý by sa dalo skúmať jednotlivé hry.

V prvom rade sme zanalyzovali fungovanie samotnej hry. Vytvorili sme si automat ťahu hráča, ktorý nám slúžil ako základ jadra simulátora. Preskúmali sme všetky karty a rozbili sme ich na základné elementy. Potom sme si vybrali, ako budeme karty reprezentovať a ako ich budeme spracovávať v programe. Tu sme sa rozhodli pre súbory xml a následné generovanie pomocou template-u. Výsledkom je, že simulátor je jednoducho rozšíriteľný o ďalšie karty, ak budú využívať implementované efekty a filtre.

Pokračovali sme implementáciou všetkých základných elementov a samotného game loop-u. Vytvorili sme API pre implementovanie hráča a potom sme sa pustili do generovania kariet, čo bola zďaleka najdlhšia časť. Pri tvorení template-u sme kde tu upravovali implementácie efektov, aby sedeli s našim cieľom.

Následne sme vytvorili konzolové rozhranie, ktoré správne vykreslovalo všetky časti hry a umožňovalo ich upravovanie. A nakoniec sme vymysleli testy, ktorými sme overili, že všetko funguje, ako má. Tým sme splnili všetko, čo sme plánovali. Jednoduchá reprezentácia kariet pomocou xml, efektívne spracovanie pomocou generácie cez template, implementácia umelého hráča pomocou nášho API a aj prehľadné užívateľské rozhranie cez konzolu.

Stavba xml súboru sa počas vývoja niekoľkokrát zmenila, keďže bolo potrebné nájsť kompromis medzi ľahkou čitateľnosťou a komplexnými efektami, ktoré sa takisto musia vygenerovať do kódu.

6.1 Možné vylepšenia

Užívateľské rozhranie je veľmi jednoduché, keďže to je konzolová aplikácia, čiže v budúcnosti by sa mohlo vylepšiť a naimplementovať napríklad cez WPF, aby bolo prehľadnejšie a aby sa dalo ovládať myšou.

Takisto by sa do takéhoto rozhrania dal pridať deck builder na stavanie balíkov. Ten by tiež z analýzi predošlích hier mohol ponúkať vylepšenia balíka. Zobrať referencie

populárnych balíkov z internetu a tie ponúkať, respektíve proti nim upozorňovať by tiež nebol zlý nápad.

7 Apendix

7.1 Apendix A

Zoznam jednotlivých efektov

- Summon - vyvolá kartu na hracie pole
- Spawn and Play - Vytvorí novú inštanciu danej karty a dá ju zahrať hráčovi
- Spawn and Summon - Vytvorí novú inštanciu danej karty a summon-ne ju
- Play Row Effect - Na daný Row aktivuje "Weather"
- Banish - Odstráni kartu z hry. Karta nejde do Graveyard-u a ani sa neaktivuje Deathwish karty
- Transform - Premení kartu na inú
- Boost - Zvýši silu karty o danú hodnotu
- Damage - Zníži silu karty o danú hodnotu
- Duel - Dve karty sa budú postupne damage-ovať až dokým jedna nebude zničená. Začína útočník. Veľkosť damage-u je sila karty.
- Destroy - Zničí danú kartu. Karta putuje do Graveyard-u.
- Seize - Nepriateľovu kartu hráč presunie sebe na hracie pole.
- Give/Remove Doomed - Pridá/Odstráni karte token Doomed.
- Give/Remove Resilience - Pridá/Odstráni karte token Resilient.
- Give/Remove Zeal - Pridá/Odstráni karte token Zeal.
- Give/Remove Immune - Pridá/Odstráni karte token Immune.
- Lock/Unlock - Zamkne/Odomkne kartu.
- Reset - Sila karty je vrátená späť na jej základnú hodnotu.
- Heal - Sila karty je zvýšená o danú hodnotu, maximálne ale do jej základnej hodnoty.
- Trigger Deathwish - Aktivuje Deathwish abilitu danej karty.
- Refresh Ability - Iba pre lídrov. Znovu sprístupní abilitu lídra.
- Shuffle Into - Zamieša kartu do danej kopy.
- Move - Presunie kartu z jedného Row na druhý.
- Repeat Deploy - Znovu aktivuje Deploy efekt karty.
- Draw - Hráč si potiahne karty.
- Consume - Karta zničí cieľ, sila cieľa je pripočítaná k sile karty.
- Discard - Karta je presunutá z ruky do Graveyard-u.

- Give/Remove Charges - Charge karty je zvýšený/znížený o daný počet.
- Clear Row Effects - Odstráni Weather z daného Row.
- Place on Top - Karta je položená na vrch balíčka.
- Set power to - Sila karty je nastavená na danú hodnotu.
- Swap power - 2 karty si vymenia sily.
- Reveal - Karta na ruke je viditeľná aj nepriateľovi.

7.2 Apendix B

Zoznam filtrov, filtruje sa z IEnumerable vstupu, na ktorý je extension metóda volaná

- FilterType - Vráti karty typu T
- FilterUnits/Artifacts/Specials - Vráti všetky Units/Artefakty/Kúzla
- FilterCategory - Vráti karty danej kategórie
- FilterFaction - Vráti karty patriace danej frakcii
- FilterCard - Vráti karty s daným menom
- FilterDamaged - Vráti karty čo majú silu menšiu ako svoju pôvodnú silu
- FilterBoosted - Vráti karty čo majú silu väčšiu ako svoju pôvodnú silu
- FilterOutImmune - Vráti karty čo nie sú Immune
- FilterPower - Vráti karty najsilnejšie alebo najslabšie
- FilterRecruit_Cost - Vráti karty najdrahšie alebo najlacnejšie
- FilterHighestBaseStrength - Vráti karty z najväčšou pôvodnou silou
- Create - Vráti 3 náhodné karty
- Top - Vráti vrchnú kartu
- Random - Vráti x náhodných kariet
- Shuffle - Vráti zamiešaný vstup
- *Nejaké extra utility metódy vo Filters.cs*
- Other - Extension na Row, vráti druhý Row
- Strength - Vráti celkovú silu vstupu

7.3 Apendix C

Zoznam a vysvetlenie jednotlivých tokenov (Flag-ov)

- Doomed - Karta s týmto tokenom je odstránená z hry namiesto toho, aby bola presunutá do Graveyard-u. Efektívne je Banished (Efekt v Apendixe A).
- Resilience - Karta s týmto tokenom nie je na konci kola odstránená z poľa, čiže je prenesená do ďalšieho kola.
- Zeal - Order efekt karty s týmto tokenom môže byť použitý v tom istom ťahu, ako bola karta zahratá. Normálne treba jeden ťah počkať.
- Immune - Karta s týmto tokenom nemôže byť vybraná hráčom ako cieľ efektu.
- Locked - Zamknutá karta má zablokované všetky jej efekty a takisto všetky ostatné aktívne tokeny.
- Spying - Token označuje kartu, ktorá bola zahraná alebo presunutá na nepriateľovu stranu poľa.

8 Zoznam Obrázkov

9 Prílohy

Gwent Simulator.zip